QNAP2 version 9.3 User's Guide

July 1996

Abstract

The Queueing Network Analysis Package, version 2 (Qnap2) is a modeling and simulation software system developed to facilitate the analysis of large and complex discrete event flow systems such as data communication networks, computer systems manufacturing facilities, and logistics systems. Qnap2 uses an object oriented representation of model components. It is comprised of a user interface language with facilities for object oriented modeling and a collection of efficient resolution algorithms, including a discrete event simulator with run length control features. The Qnap2 documentation includes the User's Guide and the Reference Manual. The User's Guide presents the features of the Qnap2 language, including the mechanisms it provides to build and analyze models. The Reference Manual describes all the language commands.

Contents

Introduction $\boxed{1}$

1.1 Overview

QNAP2 can be defined as a system for describing, handling and solving queueing network models. It is comprised of a specification language which is used for the description of the models under study and the control of their resolution, and of several resolution modules, or solvers, implementing the different techniques currently available.

1.2 Language

The language of QNAP2 allows the user to describe the following items :

The network configuration : A queueing network consists of a set of stations (one or several servers and one queue) through which circulate customers according to given routing rules; the customers may be distributed into several classes characterizing different behaviours and different processing in the stations;

The processing done by each station : The processing done by a station may be described by a simple time duration defined by its probability distribution or by a complex algorithm which may include synchronization operations;

The network resolution control : The resolution control specifies the initializations or updates of the parameters of the model under study, the activation and sequencing of the resolution methods and the editing of the results.

The QNAP2 language involves two levels :

1. a **control language**
2. an **algorithmic language** (derived from SIMULA and PASCAL) that details and completes control statements.

Syntactically, there are three kinds of control statements:

Stand alone commands: (/TERMINAL/ command, /RESTART/ command or /END/ command),

Parameterized commands: (/DECLARE/ command, /STATION/ command or /CONTROL/ command). In a QNAP2 program these commands will be followed by instructions assigning values to the parameters (default values being also generally provided). These instructions may be algorithmic blocks.

The /EXEC/ command: This command is always described by an algorithmic block.

The command language fulfills different functions :

- initialization/completion,
- model description,
- analysis control,
- solution.

Consequently, a QNAP2 program is an ordered sequence of commands. The definitions and the rules of use of these commands are to be found in the Control Language part of this manual (Control Language Chapter 3), the Algorithmic Language is defined in the first part of the manual (Algorithmic Language, Chapter 2).

1.3 Solvers

The solvers implemented in QNAP2 can be classified into four categories :

1. discrete event simulation,
   - applicable to all models,
   - high quality random number generator,
   - accuracy estimation by confidence intervals
   - transient state analysis,
2. Markov chain analyser,
   - exact results,
   - steady-state analysis,
3. exact analytical solvers:
   - analytical results based on the theorem developed by Baskett et al.,
   - convolution algorithms,
   - mean value analysis algorithms,
4. approximate analytical solvers:
   - iterative methods,
   - diffusion approximations,
   - heuristic approaches.

All these approaches yield the standard performance criteria (utilization factor, mean queue length, mean response time, throughput,...) which characterize the stationary behaviour of the network under study.

Mathematical methods may be applied only under restricted conditions. These conditions are automatically checked by QNAP2 before any treatment. Simulation is the only general resolution method, but it may be costly to use in terms of processing requirements.

The solvers are described in chapter 5, "Solvers".

1.4 Program example

Consider a network is comprised of three stations A, B and C. Each station has one FIFO server (default option). Service times at station A and C are independent exponentially distributed random variables with respective means 5.0 and TC. Station B has a constant service time equal to 3.

Example :

```
& declaration of the model elements

 /DECLARE/QUEUE A,B,C;
         INTEGER N; REAL TC;

& description of the stations

 /STATION/ NAME = A; INIT = N;
          SERVICE = EXP (5);
          TRANSIT = B, 0.2, C;

 /STATION/ NAME = B;
          SERVICE = CST (3);
          TRANSIT = A;

 /STATION/ NAME = C;
          SERVICE = EXP (TC);
          TRANSIT = A;

 & execution phase : model initialization
 & and call to the analytical solvers

 /EXEC/    BEGIN
          TC := 0.65;
          FOR N := 2, 4,8
          DO BEGIN
             SOLVE;
             PRINT("N = ", N,
                   " RESULT = ",MTHRUPUT(A));
             END;
          END;

 & Model alteration

 /STATION/ NAME = B;
          SERVICE = CST(3);

 & Control parameters of the resolution method

 /CONTROL/ TMAX = 5000; ACCURACY = A;
          MARGI = ALL QUEUE;

 & launches a new run
```

```
/EXEC/    FOR N:=2,4,8 DO SIMUL;
/END/
&
& end of the QNAP2 program
```

1.5 Fields of application

QNAP2 was initially designed as a modeling tool to carry out quantitative analyses of complex computer architectures. Although many features of QNAP2 reflect this basic motivation, QNAP2 can also be considered as a general purpose tool, packaging large segments of queueing network theory.

In this respect, QNAP2 may be used to perform modeling studies of any system which, from a structural standpoint, may be represented as a network of service stations. Examples of such systems are:

- communication systems,
- transportation sytems,
- manufacturing facilities,
- logistics facilities

Also, QNAP2 provides powerful support for theoretical work and tutorial activities in the field of queueing network theory.

1.6 Running QNAP2

The user can run the QNAP2 executable in the following way:

```
QNAP2V9 input.qnp [-o output.lis] [-l library.lib]
```

where:

- `input.qnp` is the name of the file containing the model; it is assigned to the QNAP2 predefined file `FSYSINPU`,
- `output.lis` is the name of the file into which the execution results are to be written; it is assigned to the QNAP2 predefined file `FSYSOUTP`; if no name is given, the results are printed on to the screen,
- `library.lib` is the name of the default QNAP2 library file; it is assigned to the QNAP2 predefined file `FSYSLIB`; if no name is given, there is no default library file.

Algorithmic Language $\boxed{2}$

2.1 Tokens

Tokens are the smallest meaningful units of text in a QNAP2 program. They are classified as:

- special symbols,
- reserved keywords,
- identifiers,
- labels,
- numbers,
- character strings.

2.1.1 Syntax conventions

In this manual, the QNAP2 language syntax is described using a Backus-Naur-like form:

`a → ...`	syntactic element `a` may be replaced by ...,
`[a]`	means that element `a` may be omitted,
`{ a b }`	curly braces { } are used for grouping,
`{a|b}`	means that `a` or `b` is present,
`a|b`	means that either `a` or `b` is present or nothing is present,
`a [...]`	means that `a` may be repeated,
`[a|b] [...]`	means that `a` and `b` may be repeated or omitted (e.g. `a` or `a b` or `a a b` or `b b b a`),
`a|b [...]`	means that `a` and `b` may both be repeated but at least one of them must appear,
`a [,...]`	means that a list of elements `a`, separated with commas, may be used (e.g. `a` or `a,a` or `a,a,a,a`).
`{a, b} [, ...]`	means that a list of couples `a, b` separated by commas may be used (e.g., `a, b, a, b, a, b`)

Example:

```
x → [a|b] c d|e|f [...]
```

According to the definition of `x` the following sentences are correct:

```
a c d e d e e f
c d
c d d d d
```

But the following are incorrect:

```
a c
a b c d
c d a
```

2.1.2 Character set

The character set of the language consists of:

Letter → the English alphabet letters: `A` through `Z` and `a` through `z`; the underline character _ , the numbersign #, the percent sign %, the at @ and the exclam !

Digits → the Arabic numerals `0` through `9`

Blanks → the space character and the end-of-line or return character

Special symbols

+	plus	=	equal
-	minus	<>	different
*	multiply	>	greater than
**	exponentiation	<	less than
/	divide	>=	greater than or equal to
(	opening parenthesis	<=	less than or equal to
)	closing parenthesis	:=	assignment symbol
.	period	$	macro statement symbol
,	comma	"	string separator
:	colon	&	comment separator
;	semicolon		

Note:

1. Letter case is significant: `abc` is different from `ABC`.
2. TAB characters and other control characters are not allowed in QNAP2 files.

2.1.3 Reserved words

The following words are reserved:

ALL	AND	ANY
BEGIN	DO	ELSE
END	FALSE	FOR
FORWARD	GENERIC	GOTO
IF	IN	IS
NIL	NOT	OBJECT
OR	REF	REPEAT
STEP	THEN	TRUE
UNTIL	VAR	WATCHED
WHILE	WITH	

Predefined identifiers may not be redefined by the user: predefined object types, predeclared objects, standard procedures and functions.

2.1.4 Identifiers

Syntax:

Identifier $\rightarrow$ letter [letter | digit] [...]

Semantics:

The first character of an identifier must be a letter; only the first eight characters are significant (identifiers longer than 8 characters are flagged by a WARNING). Identifiers longer than 8 characters are truncated. All the identifiers used in a QNAP2 program must be declared.

There are several reserved identifiers in the language. The complete list of these identifiers is given in Appendix D. It includes keywords (BEGIN, DO, AND...), predefined variable identifiers (FIRST, LAST, TIME...) and procedures and function identifiers (EXP, ABS, P...).

These identifiers cannot be used for declarations of items or for aliasing names.

Note:

Command names and parameter names are not reserved identifiers (STATION, SCHED...).

2.1.5 Numbers

Syntax:

integer_cst	$\rightarrow$	digit [...]
real_cst	$\rightarrow$	integer_cst[.[integer_cst]] [E[+ | −]integer_cst]

Semantics:

Numbers may be either reals or integers. Checks are made against the maximum value allowed by the host machine software.

Example:

```
0                  & integers
45
456789

45.                & reals
4E-6
3.E+4

.46 is incorrect   & but 0.46 is correct
```

2.1.6 Character strings

Syntax:

string_cst $\rightarrow$ " [character] [...] "

Semantics:

A character string is defined as a string of characters beginning and ending with the double quote character ". A character string may not contain the end of physical source line character (i.e. the two " delimiters must be on the same physical input line). If the " character is to be stored in a character string, it should be written as "".

Example:

```
" MEAN RESPONSE TIME = "
" ..............."
""                        empty string
""""                      is a character string reduced to the " character
```

2.1.7 Comments

A comment starts with the ampersand character & and includes all the characters up to the end of the physical input line.

Example:

```
& This is a comment...
```

2.1.8 Separators

The language is free format. Therefore two consecutive items have to be separated by at least one blank character or by a physical line end, except if one of the items is a special character or a comment or a character string. An identifier, a number or a keyword may not include blank characters or physical end of line.

Example:

```
/DECLARE/REAL AB, XYZ;

/EXEC/ BEGIN
         AB:= 36.12;         & is correct
         XYZ:=    AB/34E4 ;  &       "
       END;                  &       "

/EXEC/ BE GIN                & is incorrect
         AB : = 36.12;       &       "
         XYZ:= A B/34E4 ;    &       "
       EN                    &       "
       D ;                   &       "
```

Note:

The slash character "/", when used as the first non blank character of a line, introduces a QNAP2 command (refer to Chapter 3).

2.2 Types

When you declare a variable, you must state its type. A variable's type circumscribes the set of values it can have and the operations that can be performed on it. A *type declaration* specifies the identifier that denotes a type.

Data types are:

- scalar types
- array types
- object types
- reference types

Scalar types and reference types are unstructured types. Arrays and objects are structured.

Scalar types are limited to the four following types: INTEGER, REAL, BOOLEAN and STRING. New scalar types cannot be declared by the user.

There are six predefined object types:

QUEUE for queues

CUSTOMER for customers

CLASS for customer classes

EXCEPTION for exceptions (signals)

FLAG for synchronization (boolean) flags

TIMER for timers

FILE for input/output files

The user may define other object types: either independent types or sub-types of existing types.

2.2.1 Arrays

Arrays (or dimensioned variables) consist of a collection of elements of the same type (scalar, objects or references). Arrays of arrays are not permitted.

Arrays may have one or several dimensions. The number of dimensions and their ranges must be defined at declaration time (refer to section 2.3.2 "Variable and array declarations").

Example:

```
/DECLARE/

INTEGER LI (20);   & declaration of an array with
                   & 20 INTEGER entries

REAL RA (-1:3,10); & declaration of a real array
                   & having two dimensions

QUEUE DISK(10);    & declaration of an array
                   & of ten elements
                   & each element references
```

```
                         & an object of type QUEUE

INTEGER NC=3;            & declaration of an integer NC
                         & initialized to 3

CLASS X(NC);             & declaration of NC classes, with
                         & NC=3
```

Note:
The implicit enumeration rule is that the leftmost index or subscript varies the least rapidly, and the rightmost index varies the most rapidly.

2.2.2 Objects

Objects are structured variables made up of scalar elements (INTEGER, REAL, BOOLEAN or STRING), references, arrays or objects. These elements are called the object attributes. As opposed to the elements of an array, the attributes of an object may be of mixed types.

2.2.2.1 Extension of a predefined object type

All predefined object types (QUEUE, CLASS, CUSTOMER, FLAG, FILE, TIMER, EXCEPTION) have predefined attributes. A complete list of predefined attributes is given in the Reference Manual.

The user may add attributes to the QUEUE, CLASS, CUSTOMER, and FLAG objects by simply declaring them.

Example:

```
/DECLARE/

QUEUE INTEGER N;        & declaring an integer
                        & attribute of queues

CUSTOMER REAL SIZE;     & declaring two new real
CUSTOMER REAL TYPE;     & attributes of customers

OBJECT CCW;             & declaring a new object type CCW
  INTEGER ADR,DEVICE;   & declaring attributes
  REAL SEARCH(3);       & of objects of type CCW
END;
```

Each queue of the model will have an additional attribute N; each customer will have two additional attributes SIZE and TYPE. A new object type has been defined: Each CCW object will consist of two integer numbers (ADDR, DEVICE) and an array of three real numbers (SEARCH).

Note:
Several rules must be followed when adding attributes to a type:

1. objects of the type must not exist,
2. a subtype must not exist.

Example:

```
      /DECLARE/
               FILE INTEGER IFILE;    & No: predeclared files exist

               QUEUE OBJECT MACHINE;
                 STRING NAME;
               END;

               QUEUE REAL TR;         & No: a sub-type has been created
```

2.2.2.2 Definition of a new object type

The user may define other object types. An object type is defined by an object type identifier (object_type_id), by an enumeration of the attributes that are associated with this type and, optionally, by a list of formal parameters.

Example:

```
/DECLARE/
         & declaring an object type
         & identifier with formal parameters

         OBJECT NODE (N,T);
           INTEGER N;
           QUEUE CPU;        & attributes of the object type
           QUEUE DISK(N);
           REAL T;
         END;
```

Each NODE object is made of one queue CPU, N queues DISK, and one real T.

An object type may also be declared with reference to a known type (predeclared or already declared type). The new object type inherits all the attributes of the referenced type.

Example:

```
/DECLARE/ CUSTOMER OBJECT MESSAGE;
            INTEGER LENGTH;
            REAL STAT_T;
            REF QUEUE DESTINATION, ORIGIN;
          END;

/DECLARE/ QUEUE NODE;
          REF MESSAGE M;

/STATION/ NAME    = NODE;
          SERVICE = BEGIN
                      M:= NEW(MESSAGE);
                      TRANSIT (M, M.DESTINATION);
                      & ...
                    END;
```

Each object of type MESSAGE has all the attributes and the features of the type CUSTOMER, with the additional attributes LENGTH, STAT_T, DESTINATION and ORIGIN. As a consequence, an object of type MESSAGE may be used in the same fashion as an object of type CUSTOMER.

A complex hierarchy of types and sub-types may be defined. Three operators are provided to check types at run time:

- The double-colon operator :: is used to indicate that an object is in fact of a sub-type.
- The IS operator can be used to ensure that an object belongs to a specific type.
- The IN operator can be used to ensure that an object belongs to a specific type *or one of its sub-types.*

More details about these features are provided in section 2.4 “expressions”.

The fictitious type ANY can be used to declare references to objects of any type.

2.2.3 Lists

A list facility is provided to ease manipulation of objects and variables of the same type.

Example:

```
/DECLARE/

QUEUE DISK(20),CPU;
REF QUEUE Q(30);
INTEGER N=10,L(N);

/EXEC/ BEGIN
                               & list of the
         L:= 1 STEP 1 UNTIL N; & N first integers
                               &
         L:= 1 REPEAT N ;      & list of N 1s
                               &
         Q:= ALL QUEUE ;       & list of all
                               & declared queues
       END;
```

2.2.4 Data items management

Any data is created as soon as its identifier is declared.

Example:

```
/DECLARE/

INTEGER L(10);     & creation of an array
                   & of ten integer elements.
                   &
INTEGER N=12;      & creation of an integer
                   & variable with initial
                   & value of 12.
                   &
REAL R(N);         & creation of an array
```

```
                   & of 12 real elements.
```

Objects (except objects of type CUSTOMER or of any type referencing type CUSTOMER) may be created either through the static declaration facility or dynamically using the NEW function (a reference to the created object is returned as the function value).

Example:

```
/DECLARE/

& declaring an object type
& identifier with formal parameters

OBJECT NODE (N,T);
  INTEGER N;
  QUEUE CPU;
  QUEUE DISK(N);   & attributes of the object type
  REAL T;
 END;

/DECLARE/

INTEGER I;
QUEUE A;           & an object of type
                   & QUEUE is created.
REF QUEUE PROC(10);& an array of 10 references
                   & is created.
REF NODE RND;      & a reference to an object
                   & of type NODE is created

NODE (5, 10.2) ND1,ND2;
                   & static creation of 2 objects
                   & of type NODE

/EXEC/ RND:= NEW(NODE, 3, 0.5);
                   & dynamic creation of an object
                   & of type NODE

/EXEC/ FOR I:=1 STEP 1 UNTIL 10
         DO PROC(I):=NEW(QUEUE);
                   & dynamic creation of
                   & 10 objects of type QUEUE.
```

An object deletion mechanism is provided with the DISPOSE procedure.

The /RESTART/ command removes all objects.

Objects of type CUSTOMER or of any type referencing type CUSTOMER can be created only during the simulation process. As a part of the simulator they may be created:

- through a source station (station type SOURCE),
- through a static initialization inside stations (parameter INIT),
- through calls to the NEW function.

They may be destroyed:

- by the end of the simulation,
- during a simulation by a transition into the predefined OUT queue.

2.2.5 References

A reference may refer to all the objects of the same type created in a program. The type of these objects is determined at reference declaration time.

References may be used whenever an object of the corresponding type or one of its sub-types is required.

Example:

```
/DECLARE/ QUEUE CPU1,CPU2;
          REF QUEUE RQ;

          QUEUE OBJECT NODE;
            STRING NAME;
          END;

          REF NODE RN;

          NODE PARIS, NEWYORK, TOKYO, LONDON;

/EXEC/ BEGIN
         RQ:= CPU1;
         RN := LONDON;
       END;
```

When sub-types are used, the following rules apply:

1. A reference to a sub-type can be used with all the type features without restriction. This is simply of consequence of the fact that an object of a sub-type (e.g., a NODE) has all the features of the type (e.g., QUEUE).

   Example:

   `RN` can be used as a reference to a QUEUE object:

   `RN.NB`, `RN.FIRST` are valid expressions. NB and FIRST are attributes of QUEUE objects, so they are also attributes of NODE objects.

2. A reference to a type can be used with a sub-type feature, provided that:
   (a) the user explicitely specify that the reference must point at a sub-type object;
   (b) the pointed object be acceptable at run-time (i.e., be of the specified sub-type or one of its sub-types).

   Example:

   `RQ` may be used as a reference to a NODE object as follows:

   `RQ::NODE.NAME` is a legal expression because the :: operator explicitely states that `RQ` must point at a NODE or a sub-type of NODE. A run-time check is performed to ensure that this is the case.

The fictitious type ANY is used to define a reference on any object type. A REF ANY variable must always be used in conjunction with the :: operator.

Example:

```
/DECLARE/ REF ANY RA;

/EXEC/ BEGIN
         RA := LONDON;
         RA::NODE.NAME := "London City";
       END;
```

A REF ANY may point at any object, so the first statement is legal. The second one is legal only by virtue of the ::NODE.

The two operators IS and IN are provided for run-time type checking. Refer to section 2.4 "Expressions" or to the reference manual for details.

2.3 Declaration Statements

2.3.1 Overview

Syntax:

declaration_statement	→	variable_declaration	|
		attribute_declaration	|
		object_type_declaration	|
		label_declaration	|
		procedure_declaration	

Semantics:

The declaration statements must follow a /DECLARE/ command (see Chapter 3 "Control Language"). In each declaration statement one or more identifiers of variables of the same type may be declared.

Identifiers may be declared in any order. Nevertheless attributes associated with a given object type must be declared before the first object of this type is created (cf. "Data item management") (e.g. QUEUE attributes must be declared before the first queue declaration).

2.3.2 Variable and array declarations

Syntax:

variable_declaration	→	type identifier [(dim)] [=init] [,...];
type	→	`INTEGER` | `REAL` | `BOOLEAN` | `STRING`[(max_length)] |
		[`REF`] object_type_id |
		object_type_id [(actual_parameter_list)]
object_type_id	→	identifier
dim	→	bound [:bound] [,...]
bound	→	[+ | −] integer_cst | single_variable_id
single_variable_id	→	identifier
init	→	sublist
size	→	integer_cst | single_variable_id
actual_parameter_list	→	sublist [,...]

Semantics:

A string declaration may specify the maximum length of the character string, which may be given as an integer constant or an initialized integer variable. The maximum length must lie betwen 1 and 256. The default maximum string length is 80 characters.

An object declaration may include all or part of actual parameters used to initialize the object attributes (see next section for object type declarations). Sublist syntax is explained in section 2.4.5.

An array declaration specifies the number of dimensions and the size of the array together with an identifier. A dimension is specified either by an unsigned integer n specifying the number of elements in this dimension (in this case the corresponding index ranges from 1 to n), or by a pair of integers n_1:n_2 specifying the index range limits in this dimension (n_1 must be smaller than or equal to n_2). Integer constants may be replaced by initialized integer scalar variables.

The enumeration order of arrays is that the rightmost index varies the most rapidly.

Example:

```
/DECLARE/ INTEGER T (2, 3) = 1, 2, 3, 4, 5, 6;
```

yields $T(1,1) = 1$, $T(1,2) = 2$, $T(1,3) = 3$, $T(2,1) = 4$, $T(2,2) = 5$, and $T(2,3) = 6$.

Example:

```
/DECLARE/

INTEGER B (1:10);
REAL C (-1:5,2);
STRING D (0:10,2,0:3);

INTEGER A (10);

& equivalent to
& INTEGER A (1 : 10);

INTEGER N = 5;
BOOLEAN LB(N,2);

& equivalent to
& BOOLEAN LB(5,2);

STRING ST;         & size = 80
STRING(10) ST1;    & size = 10

INTEGER N=10;
STRING(N) ST2;     & size = 10

& array of 3 strings with a size = 120
STRING(120) ST3(3);

& attribute of customer with size = 30
CUSTOMER STRING(30) ST4;
```

2.3.3 Object type and attribute declarations

Syntax:

object_type_declaration	→	[object_type_ref] `OBJECT` object_type_id [(formal_parameter [,...])]; attribute_declaration [; ...] `END`;
object_type_ref	→	identifier
object_type_id	→	identifier
formal_parameter	→	identifier
attribute_declaration	→	type_id variable_declaration
type_id	→	identifier
additional_attribute_declaration	→	object_type_id type_id identifier

Semantics:

An object type declaration creates a new object type whose identifier is object_type_id. The optional object_type_ref causes the attributes of the referenced type to be inherited by the object type being declared. The attributes are declared in following declaration statements (or using the specific attribute declaration statement).
Attributes may be either:

- scalar items,
- arrays,
- references,
- objects of other known types (i.e. predefined or declared previously).

They may be classified into four types:

- normal attributes,
- parameter attributes,
- forward attributes,
- additional attributes.

2.3.3.1 Normal attribute declaration

Normal object attributes are simply declared before the END keyword. Normal attributes may be scalars, including references to other objects, or objects. The attributes of object types defined as sub-types are defined similarly.

Example:

```
/DECLARE/ INTEGER NBNODE = 10;

          OBJECT NODE;
            REAL SWITCH;
            REF NODE ROUTE (NBNODE);
            QUEUE INBUF, OUTBUF;
          END;
```

Each NODE object contains one REAL atribute, an array of 10 references to NODE objects, and two queues. The INBUF and OUTBUF queues are automatically created when a NODE object is created.

Example:

```
/DECLARE/

CUSTOMER OBJECT MESSAGE;
  REAL LENGTH;
  INTEGER LEVEL;
  REF QUEUE ORIGIN;
END;

CUSTOMER OBJECT PACKET;
  INTEGER BYTES;
  STRING HEADER;
  REF CLASS CIRCUIT;
END;
```

This example shows the declaration of two new object types MESSAGE and PACKET. These object types are both declared with reference to the predefined type CUSTOMER. Objects of these types inherit the predefined attributes and the properties of the CUSTOMER type. All operations allowed for objects of type CUSTOMER can be applied to objects of these two types.

The additional attributes are specific to the new types. They are not added to the CUSTOMER type.

2.3.3.2 Object parameters

Attributes cannot be initialized at declaration time within the object type declaration. Nevertheless, a facility is provided to allow attributes to be initialized at object creation time through a parameter mechanism.

For this purpose the formal parameter list may contain some of the attribute names which are specified in the variable declaration statements associated with the object type declaration. The list of these declaration statements follows the object type identifier and is delimited by the END keyword. At creation time (object declaration or function call NEW(object_type_id)) the actual parameter list is matched against the formal parameter list in order to make the appropriate initializations. This mechanism is provided only for scalar items, references, and arrays because objects cannot be globally assigned.

Example:

```
/DECLARE/

& declaration of a new
& object type CENTER

OBJECT CENTER (L,PROB,NLINES);

   & declaration of its
```

```
             & attributes

             INTEGER NLINES;      & number of lines
             QUEUE LINE(NLINES);  & array of NLINES queues
             REAL PROB(NLINES);   & routing probabilities
             REAL      TLINE ;    & average service time
             REF QUEUE L ;        & output queue
          END;
          ...

          /DECLARE/  & object creation

          CENTER (NIL, (0.5, 0.2, 0.2, 0.1), 4) CT;
```

Each CENTER object is composed of NLINES stations LINE with appropriate routing (array PROB and output queue L) and service (scalar TLINE) characteristics. The attributes appearing in the formal parameter list (L, PROB, NLINES) are specified when creating an object of type CENTER.

Note:
It is not mandatory that the actual parameter list length match the formal parameter list length. If the actual parameter list is shorter than the formal parameter list, the missing parameters are initialized with the default values as indicated in section 2.3.6, "Static initializations".

2.3.3.3 Forward attributes

It is possible to describe as an attribute of an object a reference on an object whose type is not already defined. A forward reference to an object is allowed only if the referenced object type declaration is performed within the same /DECLARE/ block as the object type which contains the forward reference.

Example:

```
/DECLARE/ OBJECT CONTAINER;
            REF CONTENT RCONTENT;
            .
            .
          END;

& only declarations may be placed here
& any Qnap2 command is forbidden here
& (including /DECLARE/)

          OBJECT CONTENT;
            REF CONTAINER RCONTAIN;
          END;
```

2.3.3.4 Aditional attribute declaration

The aditional attribute declaration statement is intended for adding new attributes to user-defined object types or predefined object types such as QUEUE, CLASS, FLAG and CUSTOMER.

An additional attribute is declared by writing the object type identifier followed by the attribute type identifier (for example, QUEUE INTEGER or QUEUE REAL for queue attributes of type integer or real). An attribute may be a simple variable or a dimensioned one.

Notes :

1. Attributes of a given object type must be declared before the first object of that type (or any of its sub-types) is created.
2. This feature is not (not allowed for FILE, TIMER and EXCEPTION object types as there are predeclared objects of these types.
3. Using specific attributes corresponds to an old-fashioned use of QNAP2. We recommend to avoid such declarations whenever possible because they violate object oriented design principles.

Example:

```
/DECLARE/

QUEUE INTEGER N, M;  & declaration of
QUEUE REAL STAT(5);  & QUEUE attributes.
QUEUE REAL TMIN;

OBJECT IO;           & declaration of a
 INTEGER CCW (2);    & new object type
 REF QUEUE DEVICE;   & with its
 REAL SEEK, SEARCH;  & attributes
END;

CUSTOMER REF IO CIO; & declaration of
CUSTOMER REAL TIO;   & CUSTOMER attributes
```

Each queue of the model has four additional attributes: 2 integers (N and M), a real number (TIMIN) and an array of real numbers (STAT). Each customer has two additional attributes: a reference (CIO) on IO-type objects, and a real (TIO). IO-type objects are user-defined objects. Each IO object consists of an array of integer numbers (CCW), a queue reference (DEVICE) and two real numbers (SEEK and SEARCH).

2.3.4 Object declarations

The predefined object types (object_type_id) are the following:

1. queues (QUEUE),
2. classes (CLASS),
3. flags (FLAG),
4. customers (CUSTOMER),
5. timers (TIMER),
6. exceptions (EXCEPTION),
7. files (FILE).

Example:

```
/DECLARE/

& declaration of a new
& object type CENTER

OBJECT CENTER (L,PROB,NLINES);

   & declaration of its
   & attributes

   INTEGER NLINES;     & number of lines
   QUEUE LINE(NLINES); & array of NLINES queues
   REAL PROB(NLINES);  & routing probabilities
   REAL       TLINE ;  & average service time
   REF QUEUE L ;       & output queue
END;

/DECLARE/

QUEUE CPU,DISK1,DISK2;  & creation of 3 queues
CLASS BATCH,TS;         & creation of 2 classes
QUEUE EXT;              & creation of queue EXT
FLAG SYNC;              & creating a flag
FILE F                  & creating a file

CENTER (EXT,(0.1,0.2,0.70),3) CT;
```

The last declaration statement creates an object CT of type CENTER. The attribute L references the queue EXT and the attribute PROB is an array of dimension 3 (formal parameter N) with initial values 0.1,0.2 and 0.7.

2.3.5 Reference declarations

2.3.5.1 References to objects

A reference to an object is declared by associating the keyword REF and the identifier of the object type (for instance, REF QUEUE is a reference to a queue, REF FLAG is a reference to a flag).

Example:

```
/DECLARE/

REF QUEUE PQ, LPQ(10);        & queue references
REF CUSTOMER PC,LPC(0:10,2);  & customer references
QUEUE REF QUEUE PQQ;          & queue reference attribute
REF FLAG SYNCHRO;             & flag reference
REF FILE F;                   & file reference
```

The NIL constant is the value of a reference pointing to nothing (default initial value for references).

Example:

```
/DECLARE/

QUEUE A, B;
FLAG F1, F2;
REF QUEUE Q=A;  & declaration of a queue reference
                & initially referencing A
REF FLAG F;     & declaration of a flag reference
                & referencing nothing.

/EXEC/ BEGIN
         F:=F1;           & reference F references F1
         PRINT (F.STATE); & printing the STATE
       END;               & attribute of object F1
```

2.3.5.2 Reference to any object

Syntax:
`REF ANY` identifier ;

Semantics:
ANY is a fictitious type. Such references can point to an object of any type (`QUEUE, CLASS, CUSTOMER,...`, `user type`).

Note:

1. Only references of this type can be handled, objects of this type cannot be created (statically or dynamically).
2. References to scalar variables (*INTEGER, REAL, BOOLEAN, STRING, REF*) or not allowed.

Example:

```
/DECLARE/ REF ANY RANY;
          QUEUE Q;
          CLASS CL;

/EXEC/ BEGIN
         .
         .
         RANY:= Q;
         .
         .
         RANY:= CL;
         .
         .
       END;
```

2.3.6 Static initializations

INTEGER, REAL, BOOLEAN or STRING variables may be initialized at declaration time with expressions of the same type. Similarly, references may be initialized with object expressions. These expressions are evaluated at declaration time (i.e., when the item being initialized is created). On the other hand, variables declared as attributes may not be statically initialized.

Array elements may be initialized by specifying a list of constants of the same type. This list should have, at most, as many elements as in the array. The implicit assignment rule is that the rightmost index varies the most rapidly and the leftmost index the least rapidly (cf. section 2.2.1, "Arrays or dimensioned variables").

Example:

```
/DECLARE/

INTEGER I=43, J=2;

INTEGER L(10)=(1,2,3,4,5), M=100;
REAL R(0:2,2)=(10,20,30,40,50,60);

QUEUE A,B; REF QUEUE Q=A;
```

Elements L(6),L(7),... are initialized with value 0. The elements of R are:

```
R(0,1)=10; R(0,2)=20;
R(1,1)=30; R(1,2)=40;
R(2,1)=50; R(2,1)=60;
```

The reference Q references queue A.

Unless explicitly specified, variables are initialized with the following default values:

- integers: 0
- reals: 0.
- booleans: FALSE
- strings: empty ("")
- references: NIL (empty reference)

Objects are initialized in the following way. Generally, the default attribute values for all objects are set according to the above rule.

Queues: all attributes are set to the above-listed values except for those mentioned in the description of the /STATION/ command, in Chapter 3,

Class: the attribute ICLASS is set to the rank of the class with respect to its order of creation,

Customers: all attributes are set to the above-listed values,

Flags: flags are set in the state reset (flag.STATE = FALSE),

Files: the attribute FILASSGN is an empty string and the attribute OPENMODE is set to zero.

Example:

```
/DECLARE/

INTEGER I, J, K = 4;
REAL TIMEIN,STAT(0:5)= (1,2.5,8),TIMEOUT = 2E3;
BOOLEAN OK = TRUE, BAD;
QUEUE CPU, DISK;
REF QUEUE Q = CPU, QA;
STRING MESSAGE = " WORK IS DONE ";

& The preceding initializations are equivalent
& to the following sequence of statements:

/EXEC/ BEGIN
         I:= 0;
         J:= 0;
         K:= 4;
         TIMEIN:= 0;
         STAT(0):= 1.;
         STAT(1):= 2.5;
         STAT(2):= 8.;
         STAT (3):= 0.;
         STAT (4):= 0.;
         TIMEOUT:= 2E3;
         OK:= TRUE;
         BAD:= FALSE;
         Q:= CPU;
         QA:= NIL;
         MESSAGE := " WORK IS DONE ";
       END;
```

2.3.7 Label declarations

Syntax:

label_declaration	→	`LABEL` label_id [,...];
label_id	→	identifier

Example:

```
    /DECLARE/ LABEL L1,L2;
```

2.3.8 Procedure declarations

Procedure declarations allow the use of user-coded subroutines. Three types of procedures may be declared:

- Normal procedures
- Forward procedures
- Generic procedures

2.3.8.1 Normal procedures

Syntax:

procedure_declaration	→	PROCEDURE procedure_id [(formal_parameter [,. . .])]; [local_declaration] [. . .] compound_statement;
procedure_id	→	identifier
formal_parameter	→	identifier
local_declaration	→	[VAR] variable_declaration | label_declaration

Semantics:

The formal parameter list is placed immediately after the procedure identifier. The parameters must also be declared before the description of the procedure body.

The procedure body is a compound statement, i.e., an algorithmic code sequence enclosed betwen the BEGIN and END keywords. See section 2.5, "Statements" for details.

Parameters are normally passed by value, i.e., changing the value of the formal in the procedure body has no effect on the actual parameter. Parameters may be passed by reference, when declared with the VAR keyword. In this case, changing the value of the formal parameter in the procedure body immediately affects the actual parameter.

The local variable identifiers may be used in other procedures, or as global variable, or as object attributes, without interference.

A procedure can be called within any executable algorithmic code sequence, e.g., an /EXEC/ command or the SERVICE parameter of a /STATION/ command.

Example:

```
/DECLARE/
PROCEDURE HELLO ;        & Procedure without arguments
BEGIN
  PRINT ("Hello, world!");
END;

/EXEC/ HELLO;
```

Example:

```
/DECLARE/ QUEUE Q;
          REAL SUM = 0.0,
               T = 3.0;

PROCEDURE DELAY (DELTA, D);   & procedure with arguments
VAR REAL DELTA;
REAL D;

BEGIN
  D := 2 * D;
  CST (D);
  DELTA := DELTA + D;
```

```
            END;

            /STATION/ NAME    = Q;
                      SERVICE = DELAY (SUM, T);
```

In this example, DELTA is passed by reference, and D by value. The statement `D := 2 * D` does not change the value of T. Conversely, after each service of the Q station, SUM is increased by 3.0.

Note that calling DELAY as `DELAY (5.0, T)` is a nonsense, since 5.0 is a constant. This would yield a compilation error.

Example:

```
/DECLARE/ QUEUE A,RES;

PROCEDURE REQUEST (R, Q);
REAL R;
REF QUEUE Q;

BEGIN
  P(Q);
  CST(R);
  V(Q);
END;

/STATION/ NAME    = A;
          SERVICE = REQUEST(10., RES);

& This service is equivalent
& to the following one:

/STATION/ NAME    = A;
          SERVICE = BEGIN
                      P(RES);
                      CST(10.0);
                      V(RES);
                    END;
```

2.3.8.2 Forward procedures

A forward procedure must be declared twice. The first declaration defines the name and arguments of the procedure. The second declaration defines the local variables and the procedure body.

Syntax:

```
PROCEDURE proc_name (argument_list);
& declaration of the arguments;
FORWARD;
...
PROCEDURE proc_name;
```

```
& declaration of local variables
BEGIN
...
END;
```

Semantics:

1. In the first part of the declaration, all formal parameters must be specified after the name of the procedure. After that, they must be declared between the definition of the name and the `FORWARD` keyword. Local variables may be placed also before the `FORWARD` keyword. It is highly recommended to put local variables in the second part for clarity.
2. The second part must appear in the same `/DECLARE/` block. The formal parameters must not be defined at this point. Local variables may be declared if this has not already been done. A compound statement declares the body of the procedure.

Example:

```
/DECLARE/
PROCEDURE PA (IA, RB, SC);
INTEGER IA;
REAL RB;
STRING SC;
FORWARD;

PROCEDURE PB (BD);
BOOLEAN BD;
BEGIN
  ...
  PA (1, 1.0, "1");
  ...
END;

PROCEDURE PA;
BEGIN
  ...
  PB (TRUE);
  ...
END;
```

2.3.8.3 Generic procedures

Syntax:

generic_procedure_declaration → `PROCEDURE` identifier [(formal_parameter [, ...])];
[local_declaration] ...
`GENERIC`;

The declaration of a generic procedure contains the specification of an identifier, and an optional list of parameters with the corresponding local declarations. No definition of the procedure body is authorized at this level.

A pointer on a generic procedure is defined in the same manner as a pointer on an object : the keyword `REF` is reserved for this use.

The assignment of a normal procedure to a procedure reference is performed simply with `:=`. The generic procedure and the actual procedure must have compatible formal parameter lists.

Example:

```
/DECLARE/ PROCEDURE COMBINE (X, Y, Z);
          REAL X, Y;
          VAR REAL Z;
          GENERIC;      & generic procedure combining two reals

          REF COMBINE RP;
          ...
          PROCEDURE SUMSQRT (AB, AC, BC);     & Pythagorus formula
          REAL AB, AC;
          VAR REAL BC;

          BEGIN
            BC := SQRT (AB*AB + AC * AC);
          END;
          ...

/EXEC/ RP := SUMSQRT;
```

2.3.8.4 Notes

- QNAP2 accepts only one procedure declaration level. Procedures may not be declared inside other procedures (e.g., as in Pascal).
- In case the procedure is called in the service of a virtual station, no implicit reference on the object containing the queue is available in the procedure (this is not the case in the body of the station service)
- A procedure containing customer manipulation operations should be called only within the context of simulation
- A procedure may change the value of a variable declared as a formal parameter passed by value (without VAR). This cannot, however, affect the value of the actual parameter.
- A local variable may be of any scalar type (including references).
- Local variables of a procedure may not be instances of types of predefined objects or of user types (no static creation of an object in a procedure)

2.3.9 Function declarations

Function declarations are very similar to procedure declarations. The main differences are the following:

- The FUNCTION keyword replaces the PROCEDURE keyword
- The FUNCTION keyword is preceeded with the type of the result
- Inside the body of the function, the returned result must be assigned to the predefined variable RESULT.

Example:

```
      /DECLARE/ BOOLEAN FUNCTION ISREADY (RQ);
                REF QUEUE RQ;
                REF CUSTOMER RC;
                BEGIN
                   RC := RQ.FIRST;
                   IF (RC = NIL)
                      THEN RESULT := TRUE
                      ELSE RESULT := NOT RC.BLOCKED;
                END;
```

Three types of functions may be defined, exactly like the three types of procedures:

- Normal functions
- Forward functions
- Generic functions

Refer to the previous section for details about procedure declarations.

Note:

A generic function is declared like a generic procedure. When a normal function is assigned to a reference to a generic function, the asignment statement must be followed by the ADDRESS keyword, in order to distinguish it from a call to the function.

Example:

```
      /DECLARE/ BOOLEAN FUNCTION ISOK (RQ);
                REF QUEUE RQ;
                GENERIC;

                REF ISOK RI;

                BOOLEAN FUNCTION ISUP (RQ);
                REF QUEUE RQ;
                BEGIN
                  RESULT := ...
                END;

     /EXEC/ RI := ISUP ADDRESS;       & RI := ISUP; would be incorrect
                                      & as ISUP returns a boolean
```

2.3.10 Locality

The property of *locality* exists for objects, procedures and functions. The same identifier may appear as a global variable, a local variable in one or several procedures and functions and as an attribute of one or several object types.

Example:

```
             /DECLARE/ INTEGER NUMBER;   & global variable

                       OBJECT MACHINE;
```

```
  INTEGER NUMBER; & attribute
  REAL TOPER;
END;

OBJECT CONVEYOR;
  INTEGER NUMBER; & attribute
  REAL TTRANSP;
END;

PROCEDURE WORK;
INTEGER NUMBER;   & local variable
BEGIN
  ...
END;

PROCEDURE DOIT;
INTEGER NUMBER;   & local variable
BEGIN
  ...
END;

REAL FUNCTION ISOK (NUMBER);
INTEGER NUMBER;   & (local) formal argument
BEGIN
  ...
END;
```

Note:

In old versions of QNAP2 it was possible to have an implicit reference to an attribute in a procedure called inside a SERVICE. Implicit references to attributes of predefined object types such as QUEUE or CUSTOMER or user-defined object types are no longer allowed.

2.4 Expressions

2.4.1 Overview

Syntax:

```
expression              →  simple_expression                                |
                           conditional_expression                           |
                           type_expression
simple_expression       →  comp_term [relation_operator comp_term]
conditional_expression  →  IF expression THEN expression ELSE expression
type_expression         →  object_id :: object_type_id                      |
                           object_id IS object_type_id                      |
                           object_id IN object_type_id
relation_operator       →  <|>|=|<>|<=|>=
comp_term               →  [sign] term [add_operator term][...]
sign                    →  + | −
add_operator            →  + | − | OR
term                    →  factor [mult_operator factor][...]
mult_operator           →  * | / | AND
factor                  →  primary [** primary][...]
primary                 →  constant                                         |
                           variable                                         |
                           (expression)                                     |
                           function_call                                    |
                           NOT primary
```

Semantics:

As shown in the rules for syntax, an expression is made up of constants, variables or function calls connected by logical or arithmetical operators or by parentheses. Expression evaluation is performed from left to right inside sets of parentheses according to the following operator priority rules:

- logical negation (NOT): highest priority,
- exponentiation operator ($**$),
- type operators (::, IS, IN),
- multiplication operators ($*$, /, AND),
- addition operators ($+, -$, OR),
- relational operators ($=, <>, <, >, <=, >=$).

The operators $+, -, *, /$ and $**$ apply only to primaries of type INTEGER or REAL. They yield an integer result if both operands are integers and a real result otherwise. (Note that the operation 4/5 yields 0 and 6/5 yields 1).

The operators $=$ and $<>$ may be used with any data provided that both parts of the relation have the same type. In all cases the result is boolean. The operators $<, >, <=$ and $>=$ may be used with integer, real or boolean data only (assuming that TRUE>FALSE).

Conditional expressions are composed of expressions of the same type (inside THEN clause and ELSE clause).

The :: is used to specify that a reference to an object should point at an object of a sub-type. The result of the expression is a reference to the object of the sub-type.

The IS and IN operators are used to check the type of an object. The result of the expression is a boolean.

The AND, OR and NOT operators apply only to boolean operands and yield boolean results.

An expression concerning a given object type should contain only variables of that type, including references. However, an integer expression may include real variables and conversely.

Example:

```
/DECLARE/ REAL A,B,C,D,X;
           INTEGER I;
           BOOLEAN BL;

/EXEC/     BEGIN
           C:=1.0;
           X:= A * B / C;          & (A*B)/C
           X:= A - B - C;          & (A-B)-C
           X:= A + B / C **I;      & A + (B/ (C**I))
           BL:=NOT(A<B) OR (C>D); & (NOT(A<B)) OR (C>D)

           X:= IF A>B
               THEN B              & the result is
               ELSE A;             & B or A

           A:= 3/4;                & yields A:= 0.
           A:= 3. /4;              & yields A:= 0.75
           END;
```

Note:

A > B AND C > D is incorrect since it will be interpreted as A > (B AND C) >D and (B AND C) is meaningless. The correct form is: (A>B) AND (C>B)

2.4.2 Constants

Syntax:

constant	→	integer_cst | real_cst |
		boolean_cst | string_cst |
		`NIL`
boolean_cst	→	`TRUE` | `FALSE`

Semantics:

The NIL constant is the value of an empty reference of any type (i.e., referencing nothing).

2.4.3 Variables

Syntax:

variable	→	single_variable | dimensioned_variable
single_variable	→	[simple_expression.] single_variable_id | dimensioned_variable (subscript [,...])
dimensioned_variable	→	[simple_expression.] dimensioned_variable_id
single_variable_id	→	identifier
dimensioned_variable_id	→	identifier
subscript	→	expression

Semantics:

A variable as part of an expression may be:

- prefixed with an expression having the type of the object, in the case of an object attribute,
- indexed in the case of an array.

Subscripts in a dimensioned variable must have values within the ranges defined at declaration time (reals are converted to integers according to the truncation rules). Subscript range checking is made systematically at execution time.

Each object attribute may be referenced by prefixing the attribute identifier by the identifier of the object or by a reference to it. This prefix may, in turn, be an indexed or a prefixed variable. Prefixes are evaluated from left to right.

In the instructions describing the service performed in a station (see Chapter 3, /STATION/ command, SERVICE parameter) the prefix may be omitted for queue, class or customer attributes, or attributes of objects whose type references these types. Non-prefixed queue or customer attributes are considered as refering to the current queue, to the current class or to the current customer (i.e. the customer being served).

A dimensioned variable without subscripts is used to refer to the whole array.

Example:

```
/DECLARE/ REAL X,LU(10);
          INTEGER I, CODE (5);
          STRING MESSAGE (30),S;
          CUSTOMER REAL TIMEIN;
          QUEUE REAL SIGMA;
          QUEUE CPU, DISK;
          REF CUSTOMER LC (5);

/EXEC/    BEGIN
            I:=5;
            CODE:= 1,2,3,4,5;

            & indexed variables:

            X:= LU (2);
            I:= CODE (I-4);
            S:= MESSAGE (CODE (I) + 2);
```

```
                          & prefixed variables:

                          & refers to the attribute SIGMA
                          & of the queue CPU.
                          X:= CPU.SIGMA;

                          & refers to the attribute TIMEIN
                          & of the customer referenced by the
                          & third entry of LC.
                          X:= LC(3).TIMEIN;
                        END;

          /STATION/ NAME    = CPU;
                    SERVICE = BEGIN
                                EXP (5);
                                SIGMA:= SIGMA + TIME - TIMEIN;
                              END;
```

In the preceding block the attribute SIGMA is implicitly prefixed by CPU (the current queue) and the attribute TIMEIN by a reference to the customer being served (known as CUSTOMER).

2.4.4 Function call

Syntax:

function_call	→	function_id [(sublist [,...])]
function_id	→	identifier

Semantics:

Built-in functions are used to call QNAP2 services (e.g. mathematical functions, random number generators, results, printing). Each function has a type which is the type of its result. A list of the available functions is given in the QNAP2 Reference Manual. Some functions have a variable number of parameters (e.g. GETSTAT:SERVICE:MEAN); others have no parameters (e.g. RANDU).

The user may define his own functions. When calling user functions, the number, type and order of the parameters must match the declaration of the formal arguments.

Example:

```
/DECLARE/ REAL R,X,LR(4);
          REF CUSTOMER C;
          QUEUE A; CLASS CX;

/STATION/ NAME    = A;
          SERVICE = BEGIN
                      & dynamic creation of a new customer
                      C:=NEW(CUSTOMER);
                      & ...
                    END;
```

```
/EXEC/    BEGIN
            PRINT (GETSTAT:SERVICE:MEAN (A,CX) );
            & printing results
            R:= EXP (34.);
            & generation of a random number
            X:= 1.5;
            LR:=1,2,3,4;
            & value of a discrete function
            R:=CURVE(LR,X);
          END;
```

2.4.5 Lists

Syntax:

```
list                 →  sublist [,...]
sublist              →  { simple_sublist                                    |
                          conditional_sublist} [ with_clause | repeat_clause ]
conditional_sublist  →  IF expression THEN sublist ELSE sublist
simple_sublist       →  simple_expression                                   |
                        (list)                                              |
                        dimensioned_variable [(subscript_sublist [,...])]   |
                        ALL object_type_id                                  |
                        simple_sublist.single_variable_id                   |
                        simple_expression STEP expression UNTIL expression
subscript_sublist    →  sublist
with_clause          →  WITH expression
repeat_clause        →  REPEAT expression
```

Semantics:

Lists are provided as a shorthand facility to designate and manipulate objects and variables of the same type. ALL object_type_id comprises a list of all objects of this type which have been created.

Note:

The construct ALL CUSTOMER is not allowed.

The WITH clause selects items in the specified list according to the control boolean expression; the evaluation of the boolean expression is performed for each element in the list. Control variables may be object type identifiers to allow dynamic matching of list items or list item attributes.

The REPEAT clause repeats the previous list n times, where n is the value of the integer expression (warning: the list is not reevaluated before repetition).

The STEP UNTIL clause allows a list to be built through the enumeration of the values to be generated. The control expression may be either integer or real.

Example:

```
/DECLARE/ BOOLEAN B;
          QUEUE INTEGER QI;
          QUEUE CPU, TAPE, DISK (5);
          REF QUEUE LQ(5,2);
          INTEGER XI(10);
          REF QUEUE XQ(10);

/EXEC/    BEGIN

            & sublists

            XI:= 1;               & XI(1):= 1;
            XI:= (1,2);           & XI(1):= 1 and
                                  & XI(2):= 2;
            XI:= IF B             & XI(1):= 1 and
                 THEN (1,2)       & XI(2):= 2
                 ELSE (3,4,5);    & or XI(1):= 3; XI(2):= 4;
                                  &    XI(3):= 5;
            XI:= 1 STEP 1
                   UNTIL 5;       & yields (1,2,3,4,5)

            XI:= 1 REPEAT 5;      & yields (1,1,1,1,1)

            XQ:= ALL QUEUE        & list of the objects
                 WITH NB>0 ;      & of type QUEUE having
                                  & attribute NB > 0

            XQ:= LQ(1 STEP 1      & yields LQ(1,2),..
                 UNTIL 5 , 2 );   &     ...LQ(5,2)

            XQ:= (ALL QUEUE WITH MTHRUPUT(QUEUE) > 0.5);
                                  & all declared queues
                                  & having a mean
                                  & throughput
                                  & greater than 0.5

            & lists

            XI:= 1,2;             & XI(1):=1 and
                                  & XI(2):= 2;

            XQ:= CPU,DISK;        & XQ(1):= CPU and
                                  & XQ(2):= DISK;
            XI:= 1 REPEAT 9,0;

            PRINT((ALL QUEUE).QI);
                                  & prints the value of
                                  & attribute QI for all
                                  & declared queues
          END;
```

Example:

```
/DECLARE/ REF QUEUE LQ(10);

/EXEC/    BEGIN
            & ...
            LQ:= (NEW(QUEUE)) REPEAT 10;
            & ...
          END;
```

In this example only one queue is created, and a reference to this queue is saved in each element of LQ.

2.4.6 Type operators

Syntax:

type_expression → object_id :: object_type_id |
object_id IS object_type_id |
object_id IN object_type_id

Semantics:

- The double-colon operator '::' is used to specify that a reference to an object actually references an object of a sub-type. This enables access to the attributes of the sub-type. QNAP2 performs an automatic run-time check to ensure that the referenced object has an acceptable type, i.e., is of object_type_id or one of its sub-types.
- **IS** is a type equality operator. It is used to check that a reference points at an object of a specified type.
- **IN** is a type acceptability operator. It is used to check that a reference points at an object of a specified type *or one of its sub-types.*

Example:

```
/DECLARE/CUSTOMER OBJECT MESSAGE;
                     INTEGER LENGTH;
                   END;

                   REF CUSTOMER RC;

/STATION/ NAME = ...
          SERVICE= BEGIN
                     .
                     RC::MESSAGE.LENGTH:= 10; & 10 Kbytes
                     .
                   END;
```

The :: operator enables to pass from the type CUSTOMER (RC) to the MESSAGE sub-type in order to access the LENGTH attribute. This is a kind of type casting.

```
SERVICE = BEGIN
            IF (RC IN MESSAGE) THEN PRINT (RC::MESSAGE.LENGTH);
            ...
          END;
```

The test yields TRUE if RC points at a MESSAGE object, or any sub-type of MESSAGE, which necessarily has a LENGTH attribute.

2.4.7 String operator

concatenation	→	string_expression // string_expression	
string_expression	→	string_cst	|
		string_function_call	|
		string_expression	

The concatenation operator // may be used several times in a parenthesis. Function calls are possible as for other operators. If the dimension of the result of the concatenation exceeds the size of the destination variable, it is truncated.

Example:

```
/DECLARE/ STRING(10) S1,S2,S3;

/EXEC/ BEGIN
         PRINT("Execution of the program produces:");
         S1 := "ABCDEFG" // "HIJKL";
         PRINT("R1 : ",S1);
         S1 := "ABC";
         S2 := " DEF";
         PRINT("R2 : ",S1//S2);
       END;
```

Execution of the program produces:

```
R1 : ABCDEFGHIJ
R2 : ABC DEF
```

Note the truncation in the first result.

Note:
Be careful not to place the // operator as the first element on an input line. This will result in a syntax error, as the / character introduces QNAP2 commands when used as the first non-blank character.

2.5 Statements

2.5.1 Simple statement and compound statement

Syntax:

statement	→	[simple_statement | compound_statement]
compound_statement	→	`BEGIN`
		{ [label_id: [. . .]] [statement [; . . .] } [. . .]
		`END`
simple_statement	→	assignment_statement |
		goto_statement |
		procedure_call |
		if_statement |
		for_statement |
		while_statement

Semantics:

Any instruction may be labelled, provided that it is enclosed in a BEGIN . . . END block. The identifier used must have been declared as a LABEL.

Note:

A statement may be empty. ';;' is accepted inside a compound statement only.

2.5.2 Assignment statement

Syntax:

assignment_statement	→	single_variable `:=` expression |
		dimensioned_variable `:=` list

Semantics:

Variables and expressions used in an assignment should be of the same type. However integer and real numbers may be mixed. If a real expression is assigned to an integer variable the value assigned is the integer part of the real number rounded towards zero.

In the case of references, the expressions may contain only references of the same type (i.e., referencing objects of the same type).

In the case of an assignment with a dimensioned variable, QNAP2 checks that the list length is less than or equal to the dimensioned variable length. Trailing unassigned elements are not modified. The assignment is made according to the enumeration rule specified in the section 2.2.1, "Arrays or dimensioned variables."

Example:

```
&
& declarations
&
/DECLARE/ INTEGER LI (20); REAL R; BOOLEAN B1, B2;
          REF QUEUE PQ;
          QUEUE INTEGER AI;
```

```
                    QUEUE A;
                    REF CUSTOMER C;
                    REF QUEUE DISK(10);
          &
          &         assignments
          &
          /EXEC/    BEGIN
                      LI (1):= 123;
                      R:= 23.4;
                      B1:= FALSE;
                      PQ:= A;
                      B2:= LI (2)>= 3;
                      C:=C.NEXT;
                      PQ.AI:= 36;
                      PQ:= IF AI=0 THEN A ELSE DISK(LI(1));
                      LI:=0;
                      LI:=1 STEP 1 UNTIL 10;
                    END;
```

Note:

Conversion rules may induce problems that should be emphasized.

Example:

```
/DECLARE/ REAL U; INTEGER I;

/EXEC/    BEGIN
             U:= 4;      & U = 4.0
             U:= 3/4;    & U = 0.0
             U:= 3./4;   & U = 0.75
             I:= 2.37;   & I = 2
             I:= 2.999;  & I = 2
          END;
```

Note:

Forgetting the array index is a common trap for QNAP2 beginners:

```
/DECLARE/ REAL T (10);
          REAL U;

/EXEC/ T := 2.0;          & Yields T (1) = 2.0
                          & but...
/EXEC/ U := T;            & yields an error message
```

2.5.3 Procedure call

Syntax:

procedure_call → procedure_id [(sublist [,...])]

Semantics:

Procedures are either QNAP2 system routines (e.g., resolution algorithms, synchronization mechanisms, printing) or user-provided routines. A list of the QNAP2 procedures is given in the QNAP2 Reference Manual.

Some procedures accept a variable number of parameters (e.g., PRINT, NETWORK). Others have no parameter (e.g., SIMUL).

User-defined procedures must be declared and described inside a declaration statement. When calling user procedures, the number, type and order of the parameters must match the declaration of the formal arguments.

Example:

```
/DECLARE/ CLASS X;
          QUEUE REAL TETA;
          QUEUE CPU,A;

PROCEDURE DELAY(T);
REAL T;
BEGIN
  CST(T);
  PRINT(TIME);
END;

/STATION/ NAME    = CPU;
          SERVICE = BEGIN
                      EXP (23);
                      P(A);
                      DELAY(TETA);
                      TRANSIT (CUSTOMER, A, X);
                    END;
```

2.5.4 GOTO statement

Syntax:

goto_statement → `GOTO` label_id

Semantics:

Execution of a GOTO statement causes an unconditional transfer to the statement labelled label_id. Labels have to be declared identifiers (with the LABEL type). The labelled statement and the GOTO statement must be part of the same compound statement (in a SERVICE parameter, a TEST parameter or an EXEC command). All statements may be labelled, provided they are parts of a compound statement.

Example:

```
/DECLARE/ LABEL LA,LB,LC;
          INTEGER I;

/EXEC/    BEGIN
```

```
                 IF I<3 THEN GOTO LA ELSE GOTO LB;
                 LA: I:= I+2; GOTO LC;
                 LB: I:= 0;
                 LC: PRINT (I);
               END;

               & error
     /EXEC/    IF I>0 THEN GOTO LA;
```

2.5.5 IF statement

Syntax:

if_statement → `IF` expression `THEN` statement [`ELSE` statement]

Semantics:

Execution of an IF statement causes the evaluation of the boolean expression following the keyword IF. If the expression is TRUE, execution continues with the statement in the THEN clause; otherwise, execution continues with the statement in the ELSE clause, if present, or with the statement following the IF statement itself if the ELSE clause is not present. Up to 10 levels of IF statements may be nested.
The following statement:

```
     IF expr1 THEN IF expr2 THEN stmt1 ELSE stmt2 ;
```

is interpreted as:

```
     IF expr1
        THEN BEGIN
               IF expr2 THEN stmt1 ELSE stmt2
             END;
```

Example:

```
     /DECLARE/ CLASS BATCH;
               INTEGER N;

     /EXEC/    BEGIN
                 IF (CLASS = BATCH)
                    THEN BEGIN
                           N := N + 1;
                           IF (N > 20) THEN N := 0;
                         END
                    ELSE N := N - 1;
               END;
```

In the preceding example the syntax would have been incorrect if the END had been followed by ";". This rule is a common trap for QNAP2 beginners.

2.5.6 WHILE statement

Syntax:

while_statement → `WHILE` expression `DO` statement

Semantics:

Execution of a WHILE statement causes the repeated execution of the statement following the DO clause while the boolean expression remains TRUE. The boolean expression is evaluated before each execution of the statement. If the expression is initially FALSE the statement is not executed.

Example:

```
/DECLARE/ INTEGER N;
          REAL S,S2,X;
          BOOLEAN BL=TRUE;

/EXEC/    BEGIN
            WHILE (N <= 8) DO
              BEGIN
                N:= N+1;
                S:= S + X;
                S2:= S2 + X*X;
              END;
            WHILE BL DO
              BEGIN
                N:=N-1;
                BL:=FALSE;
              END;
          END;
```

2.5.7 FOR statement

Syntax:

for_statement → `FOR` single_variable:= list `DO` statement

Semantics:

Execution of a FOR statement causes the repeated execution of the statement following the DO clause according to the values of a control variable. The values of the control variable may be specified by means of a list of expressions of the same type. All the list features are allowed for assigning the control value, i.e., STEP UNTIL clause, REPEAT clause, WITH clause, etc.

The list of expressions is evaluated when entering the FOR loop and is not reevaluated at each execution of the loop. If the control variable is indexed or prefixed then the index and prefix are also evaluated only once, when entering the FOR statement.

Example:

```
/DECLARE/ INTEGER I, N; REAL X, L (100);
```

```
                QUEUE A, B, C; REF QUEUE Q;

      /EXEC/    BEGIN
                  FOR N:= 1 STEP 1 UNTIL 100 DO
                      X:= X + L (N);

                  FOR N:= 100 STEP -1 UNTIL 1
                      DO X:= X + L (N);

                  FOR Q:= A, B, C DO
                      BEGIN
                        PRINT ("QUEUE:",Q,"L = ",MCUSTNB (Q));
                        PRINT ("THRUPUT ", MTHRUPUT (Q) );
                      END;

                  FOR I:=0, 10, 100 STEP 100 UNTIL 1000 DO
                      PRINT(I);

                  FOR Q:= ALL QUEUE WITH MTHRUPUT(QUEUE) > 0. DO
                      PRINT (Q,MTHRUPUT(Q));
                END;
```

2.5.8 WITH statement

Syntax:

with_statement → `WITH` object_id `DO` statement

Semantics:

`WITH` provides an implicit reference to the object being manipulated. Its use is allowed in any algorithmic block.

Example:

```
      /DECLARE/ OBJECT MESSAGE;
                  INTEGER LENGTH;
                  STRING TYPMESS;
                END;

                MESSAGE MESS1;

      /EXEC/ WITH MESS1 DO
               BEGIN
                 LENGTH :=GET(INTEGER);

                 & equivalent to:
                 & MESS1.LENGTH:=GET(INTEGER);

                 TYPMESS:=GET(STRING);
```

```
  & equivalent to:
  & MESS1.TYPMESS:=GET(STRING);
END;
```

2.6 Input/output facilities

The algorithmic language of QNAP2 provides a set of predefined procedures and functions for input/output. The I/O facilities can be classified into three categories:

1. Reading/writing data from/to text files or to the terminal.
2. Saving/restoring a complete model to/from a library file.
3. Plotting charts on a graphical terminal, printer or plotter.

2.6.1 Data files

QNAP2 data files use the standard Fortran 77 *formatted* input/output features. The data files are standard human-readable files.

Input is performed with the GET and GETLN functions. Output is performed with the PRINT, WRITE, and WRITELN procedures.

Additionally, a number of procedures available are to open, close and perform other control operations on data files.

2.6.1.1 Input functions

Input files may contain the following items:

- integer numbers;
- real numbers;
- boolean values;
- strings;
- object identifiers;
- comments.

Data files must follow syntax rules similar to those of QNAP2 source files: free format, no tab or other control characters. Data items should be separated by blank characters or end of line. Blank lines and comments are ignored (except when explicitely skipped with GETLN).

BOOLEAN values must explicitely be written as `TRUE` or `FALSE`.

Object identifiers must have been declared before attempting to read them.

Example:

```
"Bristol Factory - First design"

& Control parameters:

100.0    10000.0           & TSTART and TMAX
TRUE                       & detailed stats

& Production plan:

5000                       & number of items of each product class
3500
2900

& Failure analysis:
```

```
CONVEYOR                    & failing device
5000.0                      & failure date
```

In this example, CONVEYOR is an object identifier (e.g., a QUEUE) which must have been declared before being read.

Syntax:

```
{GET | GETLN} ([file,] type_id [, length])
```

Semantics:

The GET and GETLN functions work in a very similar way. The only difference is that GETLN skips to next line after reading, thus ignoring the rest of the line.

GET is used to read one data item at a time.

The optional `file` is used to specify where to get the data. The default is the predefined file FSYSGET. `type_id` specifies the type of the data item: INTEGER, REAL, BOOLEAN, STRING or a previously declared object type.

The optional `length` argument is the number of characters to read. It should be used only to read unquoted strings, as all other data items are naturally delimited, and the input format is normally free.

Example:

```
/DECLARE/ INTEGER ITEMS = 3;
          INTEGER I, COUNT (ITEMS);
          STRING TITLE, IDENT (ITEMS);

/EXEC/ BEGIN
          TITLE := GET (STRING);
          FOR I := 1 STEP 1 UNTIL ITEMS DO BEGIN
              IDENT (I) := GET (STRING, 8);
              COUNT (I) := GETLN (INTEGER);
          END;
       END;
```

The first GET reads a quoted string. The GET calls read 8 character strings. The GETLN calls read free format integers and skip to the next line. The data file should look like:

```
"Data for Bristol Factory - Design 1"
PABC01XY    1000
PABC01XZ    1250
PABC01ZT    1570
```

2.6.1.2 Output procedures

Syntax:

```
PRINT | WRITE | WRITELN [ ([file,] {data_item [, format]} [, ...] ) ]
```

Semantics:

PRINT, WRITE and WRITELN work in a very similar way. The main differences are the following:

- The default format used by PRINT is a fixed length format for all data types (except strings), whereas WRITE and WRITELN use a variable length format.
- PRINT and WRITELN skip to the next line after printing, whereas WRITE stays on the same line.
- PRINT always inserts a blank character at the beginning of the output line (Fortran 77 standard). With WRITE and WRITELN, this is left under the user's responsibility.

The `file` argument specifies the destination of the output. The default destination is FSYSPRINT. By default, it is identical to the QNAP2 standard output FSYSOUTPUT.

The optional `format` argument specifies the output format with the following syntax:

- `: width`, or
- `: width : decimal_places` (real numbers only)

`width` specifies the total number of characters to print the data item. `decimal_places` specifies the number of digits to print after the decimal point.

Example:

```
WRITELN ("Result:" : 10, N : 10, X : 10 : 4);
```

The default format used by PRINT is the following:

Type	Width	Details
INTEGER	10	sign, 8 digits, 1 space
REAL	12	12 characters, 4 significant digits
BOOLEAN	4/5	TRUE or FALSE explicitely printed
STRING		actual string length
Object	10	8 characters, 2 spaces

The actual format for reals depends on the magnitude of the number. Very small and very large numbers are printed with the scientific mantissa/exponent notation (e.g., 1.057E+9).

The default format for WRITE and WRITELN is to print only the minimum number of characters (no surrounding spaces). Reals are printed with four significant digits. Strings are right justified.

Note:

1. `WRITELN;` and `PRINT ("")` both print a blank line
2. `PRINT;` prints a "1" in the first column. This corresponds to a form-feed in the Fortran 77 standard.
3. Beware of short formats: if the data item does not fit into the specified width, QNAP2 raises an error condition.

2.6.1.3 Control procedures

The following procedures are available to control data files:

FILASSIGN is used to assign a file name to a QNAP2 FILE object.

OPEN and CLOSE are used to open and close files.

SETSYN is used to define a *synonym* file (I/O redirection).

FILSETERR is used to set the error recovery level for I/O operations.

SETRETRY is used to set the number of read attempts allowed before raising an error condition.

SETBUF is used to set the I/O buffer size of the FILE object, which should match the file's record length.

Only FILASSIGN, OPEN and CLOSE are described here. For the other procedures, please refer to the Reference Manual.

Syntax:

```
FILASSIGN (file, name)
OPEN (file [, mode])
CLOSE (file [, mode)
```

QNAP2 FILE objects must connected to a physical file. This is performed with FILASSIGN. Once the file name assigned to the FILE object, all subsequent operations are performed via the FILE object. This ensure model portability, as the file names are operating system dependent.

The optional `mode` argument of OPEN can take the following values:

1 to open a file for reading. The file must already exist.

2 to create a new file. The file must not exist.

3 to open a file for writing. The file may already exist.

Note:

1. Most predeclared files are assigned through the QNAP2 startup procedure. It is not required to them before using them: some of them are already opened, or an automatic OPEN is performed in the right mode.
2. Attempting to open a file that is already opened, or to close a file that is already closed, yields an execution error.
3. FILASSIGN sets the FILE attribute FILASSGN.
4. OPEN and CLOSE set the FILE attribute OPENMODE.
5. When an error occurs during any I/O operation, QNAP2 sets the FILE attributes ERRSTATUS and HARDIOST.

2.6.2 Library files

Library files can be used to save a complete model for later processing. A library file contains an image of QNAP2's working memory. It contains all the declared object types, variables, procedures and objects declared or created at the time the model was saved.

Library files can be used to save compilation time when a model is run frequently. They can be used to save a model for later debugging, or to freeze the model state during simulation and restart from the frozen state.

Syntax:

```
SAVE | SAVERUN ([file,] label)
RESTORE [ ([file,] label) ]
```

Semantics:

The SAVE and SAVERUN procedures are used to save the model in a library file. The default `file` is FSYSLIB. It is normally assigned to a physical file through QNAP2 startup procedure. If another file is used, it should be assigned with FILASSIGN before saving the model. The `label` argument is a string that is stored in the header of the library file.

SAVE can be used only within an /EXEC/ block. SAVERUN can be used in any algorithmic code sequence, including during simulation.

The RESTORE procedure is used to restore the model from the library file. The default `file` is FSYSLIB. The optional `label` argument is compared against the one stored in the library file. If the labels do not match, QNAP2 prints an error message and does not restore the model.

When the model has been saved with SAVE, the statements following the RESTORE are skipped until the next control language command.

When the model has been saved with SAVERUN, the /REBOOT/ command can be used to manipulate the model state before resuming the execution (see chapter 3, "Control Language").

Example:

The first model declares a number of procedures, functions and variables. All the compiled code is saved in the default library file:

```
/DECLARE/ ...

PROCEDURE READDATA;        & Read data file and initialize model
BEGIN
  ...
END;

/CONTROL/ ENTRY = READDATA; & Call READDATA just before simulation

/EXEC/ SAVE ("DECLARATIONS");
/END/
```

The second model is just three lines long: it restores all the declarations, and launches the simulation:

```
/EXEC/ RESTORE;
/EXEC/ SIMUL;
/END/
```

Note:

1. As opposed to data files, library files are binary files. They are not human-readable, but the resulting format is much more compact that what is possible with formatted text files.
2. Library files should not be opened or closed by the user. This is performed automatically by QNAP2 when required.

Control Language $\boxed{3}$

3.1 Control Statements

3.1.1 Control language overview

Syntax:

program	→	command [...] /END/
command	→	declare_command |
		station_command |
		control_command |
		terminal_command |
		exec_command |
		reboot_command |
		restart_command

Semantics:

A QNAP2 program consists of different functional parts introduced by QNAP2 control statements: model initialization part and solution part.

The control language is similar to the algorithmic language (free format, similar items). It uses an additional set of keywords (command names, command parameters and options). These keywords are not reserved words nor predefined identifiers for the algorithmic language (e.g. STATION, NAME, FIFO, TMAX).

Note:

Only the first four characters of a keyword are significant.

Each command is introduced by a keyword enclosed with two slash characters (/.../). The first slash of the command string must be the first non blank character of a line. Therefore only one command may appear on a line.

QNAP2 commands are:

`/DECLARE/`	identifiers declaration,
`/STATION/`	work station characteristics description,
`/CONTROL/`	QNAP2 control parameters specification,
`/EXEC/`	algorithmic instructions block execution,
`/TERMINAL/`	QNAP2 program interactive execution,
`/REBOOT/`	algorithmic reboot block after a deferred RESTORE,
`/RESTART/`	introduction of a new model,
`/END/`	end of QNAP2 source file.

Example:

Definition of a station and of a control parameter

```
/DECLARE/ QUEUE A,B;

/STATION/ NAME = A;
          INIT=1;       & number of customers in A

/CONTROL/ TMAX = 1000.; & simulation duration
/EXEC/    SIMUL;        & simulation with
                        & one customer only.
```

```
/STATION/ NAME = A;
          INIT = 3;
/EXEC/    SIMUL;        & simulation with
                        & 3 customers.

/CONTROL/ TMAX = 5000.;
/EXEC/    SIMUL;        & simulation with
                        & 3 customers
                        & duration limited to 5000
```

The preceding example can also be written as shown below by associating a variable N to the number of customers in A and a variable T to the simulation duration:

Example:

```
/DECLARE/ QUEUE A,B;
          REAL T; INTEGER N;

/STATION/ NAME = A; INIT = N;

/CONTROL/ TMAX = T;

/EXEC/    BEGIN
            N:= 1;
            T:= 1000.;
            SIMUL;
            N:= 3;
            SIMUL;
            T:= 5000.;
            SIMUL;
          END;
```

3.1.2 Command parameters evaluation

During a session the QNAP2 system enters different processing phases according to the different commands encountered in a QNAP2 program. A definition of these phases is necessary for a clear understanding of the processing done by QNAP2 during each of them and the way the expressions and the commands appearing in a QNAP2 program are evaluated. The following phases are defined:

compile time: compile time corresponds to the compilation of QNAP2 commands. A compile time phase terminates when the statement associated with the first encountered /EXEC/ command has been compiled.

execution time: execution time follows compile time. It corresponds to the execution of the statement associated with the last compiled /EXEC/ command.

initiation time: initiation time starts when a resolution procedure is called during execution time. During initiation time the called solver performs various operations in order to retrieve and check the data necessary for the analysis. For example, the expressions appearing in the right-hand side of the TRANSIT parameters of a /STATION/ command are evaluated only once at initiation time.

solution time: solution time corresponds to the analysis of the model by the requested solver. For example, the expressions appearing in the right-hand side of the parameters SERVICE of a /STATION/ command are evaluated at solution time.

The phase during which the expressions appearing in the parameters of the commands of a QNAP2 program are evaluated is specified in the next sections. The details of the operations performed during initiation time and solution time are given in Chapter 5.

3.1.3 Syntactic conventions

Syntax:

integer	→	expression
real	→	expression
queue	→	expression
class	→	expression
file	→	expression
exception	→	expression
timer	→	expression
integer_sublist	→	sublist
real_sublist	→	sublist
queue_sublist	→	sublist
class_sublist	→	sublist
file_sublist	→	sublist
exception_sublist	→	sublist
timer_sublist	→	sublist
integer_list	→	list
real_list	→	list
queue_list	→	list
class_list	→	list
file_list	→	list
exception_list	→	list
timer_list	→	list

Semantics:

If an integer, a real, a string or a reference to an object is needed in a command language syntax definition, a constant or an expression of the same type may be used as well.

3.2 /DECLARE/ Command

Syntax:

declare_command → `/DECLARE/` [declaration_statement] [. . .]

Semantics:

The purpose of the /DECLARE/ command is to introduce a list of declarations for the identifiers used in a QNAP2 program. This command must be followed by the declaration statements whose syntax is described in the corresponding paragraph in Chapter 2.

The /DECLARE/ command may occur several times in a program (if, for instance, new identifiers must be declared). All the identifiers have to be declared before being first encountered and may not be redeclared, except after a /RESTART/ command.

The evaluation of the expressions appearing in a /DECLARE/ command is done at compile time.

Example:

```
/DECLARE/QUEUE A,B;

/STATION/ NAME    = A;
          TRANSIT = B;
          SERVICE = EXP (2);

/STATION/ NAME    = B;
          TRANSIT = A;
          SERVICE = EXP (5);

/DECLARE/ INTEGER N; & new declarations
          REAL T;

/STATION/ NAME = A;  & alteration
          INIT = N;  & of station A

/CONTROL/ TMAX = T;
```

3.3 /EXEC/ Command

Syntax:

exec_command $\rightarrow$ /EXEC/ statement ;

Semantics:

The purpose of the /EXEC/ command is to introduce and execute a statement of the algorithmic language (simple statement or compound statement). This statement is intended to bring the real analysis into play and perform the following functions:

- initialization of variables,
- call to resolution procedures,
- display of results.

The statement associated with an /EXEC/ command is executed at execution time, i.e. before the compilation of the next command in the program and immediately after the compilation of this /EXEC/ command.

Example:

```
 /DECLARE/ INTEGER N;
           QUEUE A,B;

/STATION/ NAME    = A;
          INIT    = N;
          TRANSIT = B;
          SERVICE = EXP (1);

/STATION/ NAME    = B;
          TRANSIT = A;
          SERVICE = CST (2);

/CONTROL/ TMAX     = 1000.;
          ACCURACY = ALL QUEUE;

/EXEC/    BEGIN
            N:= 1;
            SIMUL;
            PRINT(" RESPONSE ", MRESPONSE(A) + MRESPONSE(B));
          END;

& execution of the preceding block:
& one simulation run.

/STATION/ NAME    = B;
          TRANSIT = A;
          SERVICE = CST (3);

/EXEC/    FOR N:= 1 STEP 1 UNTIL 5 DO SIMUL;
```

```
& execution of the previous intruction:
& 5 simulation runs.
```

3.4 /TERMINAL/ Command

Syntax:

terminal_command → /TERMINAL/

Semantics:

The /TERMINAL/ command allows interactive work with QNAP2, provided that QNAP2 runs on a timesharing system. Thus a whole QNAP2 program, or part of it, may be input and executed interactively.

Example:

The first part of the program describing the model under study is in a file assigned as QNAP2 input. The commands controlling the analysis of the model are read from the terminal.

```
&         model description

/STATION/ NAME = A;

&         the remaining part of the
&         program is read
&         from the terminal ...

/TERMINAL/

& terminal input:

&         an analytic solution of
&         the model is requested

/EXEC/    SOLVE ;

&         station A is modified

/STATION/ NAME    = A;
          SERVICE = EXP (5);

&         a new analysis
&         is requested

/EXEC/    SOLVE ;
```

Note:

Use of the /TERMINAL/ command should be restricted to very short uses. As it is not possible to change a previous input line, typing errors are sometimes un-recoverable.

3.5 /RESTART/ Command

Syntax:

restart_command → `/RESTART/`

Semantics:

The /RESTART/ command is used to restore the initial state of QNAP2. It clears the data and the workspace associated with the current model and it starts the analysis of a new independent model in the same program on the default FORTRAN unit. No declaration nor result is transmitted from the previous model; in order to work with submodels sharing data and results, the NETWORK procedure should be used.

Example:

```
& model 1

/DECLARE/ QUEUE A, B, C;
          CLASS X, Y;

/STATION/ NAME = A;

/EXEC/ SIMUL;

/RESTART/

& model 2:

/DECLARE/ QUEUE A, B, C, D;

/EXEC/ SOLVE ;

/END/ & end of the physical program
```

3.6 /REBOOT/ Command

Syntax:

reboot_command → `/REBOOT/` statement ;

Semantics:

The /REBOOT/ command can be used immediately after a *deferred* RESTORE procedure call. The reboot statement can be used to perform some initializations before resuming the execution at the point where the SAVERUN was performed.

The reboot statement may include any algorithmic language operation that was allowed at the point where the SAVERUN was performed. This includes customer manipulation and settings of simulation options (e.g., TRACE, TMAX) if the SAVERUN was performed during simulation.

This feature has several interesting applications. For example:

- Debugging: when an error condition is detected, QNAP2 can (optionally) save the model for later debugging. The user may also detect error conditions and save the model.
- Variant analysis: A first simulation is performed until some condition is reached (e.g., steady-state, component failure...) and the model is saved. The simulation can be resumed from the save point with different options.
- Interactive simulation control: the simulation runs until some condition is reached (e.g., decision point) and the model is saved. The user is then presented the system state and prompted for a decision. Then the simulation goes on.

Example:

```
& First simulation model:

/STATION/ NAME = A;
          SERVICE = BEGIN
            IF (NB > 10) THEN BEGIN
               PRINT ("WARNING: Too many customers");
               SAVERUN ("TOOMANY");
            END;
            ...
          END;
...
/END/

& Second simulation model:

/EXEC/ RESTORE ("TOOMANY", "DEFERRED");

/REBOOT/ BEGIN          & /REBOOT/ must follow immediately the RESTORE
           PRINT ("Simulation stopped at ", TIME);
           HALT;        & call the interactive debugger
         END;
 ...
```

3.7 /STATION/ Command

3.7.1 Overview

Syntax:

station_command	→	`/STATION/` name_parameter [station_parameter][...]
station_parameter	→	type_parameter |
		copy_parameter |
		service_parameter |
		sched_parameter |
		init_parameter |
		prior_parameter |
		rate_parameter |
		quantum_parameter |
		transition_parameter |
		split_parameter |
		match_parameter |
		fission_parameter |
		fusion_parameter

Semantics:

The function of the /STATION/ command is to describe one or several stations of the model. It may be used for the initial definition of a list of stations as well as for the subsequent alterations of their characteristics.

The parameters of the /STATION/ command are:

`NAME`	:	queue identifiers,
`TYPE`	:	station type,
`SCHED`	:	customer scheduling,
`INIT`	:	initial state of the queue,
`SERVICE`	:	service description,
`TRANSIT`	:	customer transitions,
`SPLIT`	:	splitting and transition of customers,
`MATCH`	:	joining up of split customers,
`FISSION`	:	fission and transition of customers,
`FUSION`	:	fusion of customers,
`RATE`	:	service rate,
`CAPACITY`	:	maximum queue capacity,
`REJECT`	:	rejection statement for customers in excess,
`PRIOR`	:	priority level of customers,
`QUANTUM`	:	service quantum allocated to customers,
`COPY`	:	parameters copied from another station.

The syntax rules of these parameters are given below.

3.7.2 NAME parameter

Syntax:

name_parameter $\rightarrow$ `NAME` = { queue_list | $*$ object_type_id | $*$ object_type_ididentifier} [,...] ;

Semantics:

The NAME parameter must be the first parameter of the /STATION/ command. The command defines as many stations as there are queues in the queue list, each station being identified by the name of its associated queue (there is only one queue per station, two stations may not share a queue). The $*$ notation is used to describe a template queue. This facility may be used in the two following cases:

1. description of objects defined as sub-types of QUEUE ($*$ followed by the object type identifier);
2. description of queues defined as attributes of an object ($*$ followed by the object type identifier, a dot and the attribute identifier.

The queue list expression is evaluated at compile time.

Example:

```
/DECLARE/ QUEUE CPU,DK(10);

& definition of the stations associated
& with the five first queues of DK(10)

/STATION/ NAME    = CPU;
          TRANSIT = DK, (1 REPEAT 5, 1 REPEAT 5);
          SERVICE = EXP (1);

/STATION/ NAME    = DK(1 STEP 1 UNTIL 5) ;
          TRANSIT = CPU;
          SERVICE = EXP(0.1);

& alteration of the previously
& defined CPU station

/STATION/ NAME = CPU;
          INIT = 5;
```

Example:

```
/DECLARE/ INTEGER N = 10;
          QUEUE REAL X, PROB(N);
          QUEUE INTEGER I;
          QUEUE Q(N);

/STATION/ NAME    = Q;
          INIT    = I;
          TRANSIT = Q, PROB;
          SERVICE = EXP(X);
```

This simple model description defines a general closed network of N stations. The mean service time, the initial number of customer and the transition probabilities of each station are attributes of the queue associated with this station.

Example:

```
/DECLARE/ QUEUE OBJECT LINE;
                 INTEGER SPEED;
                END;

          OBJECT NODE;
                   REAL TNODE;
                   QUEUE CPU, BUFFER;
                 END;

&         template station LINE:

/STATION/ NAME    = * LINE;
          SERVICE = EXP(1. / QUEUE::LINE.SPEED);

&         template station BUFFER:

/STATION/ NAME    = * NODE.BUFFER;
          SERVICE = EXP (TNODE);
          TRANSIT = CPU;
```

Note that in the description of the BUFFER station, an implicit reference on the including NODE object is automatically provided. This is not the case for the LINE object. The predefined variable QUEUE references the current queue. It must explicitely be specified as a LINE object with the :: operator in order to access the SPEED attribute.

3.7.3 TYPE parameter

Syntax:

type_parameter	→	`TYPE` = station_type;	
station_type	→	`SERVER` [,multiple]	|
		`RESOURCE` [,multiple]	|
		`SEMAPHORE` [,multiple]	|
		`SOURCE`	|
		multiple	
multiple	→	`SINGLE`	|
		`MULTIPLE` (integer)	|
		`INFINITE`	

Semantics:

The parameters have the following meanings:

SERVER: defines a server station; the number of servers is specified by:

`SINGLE`	the station is a single server,
`MULTIPLE(n)`	the station is made of n identical servers ($n >= 0$),
`INFINITE`	the station is made of an infinity of servers (hence it is a pure delay),

RESOURCE: defines a resource station; the number of resource units available is specified by:

`SINGLE`	the resource may not be shared,
`MULTIPLE(n)`	the resource may be shared by n customers,
`INFINITE`	the resource is always available,

SEMAPHORE: defines a semaphore station; the initial value of the counter is specified by:

`SINGLE`	the initial value of the semaphore counter is 1,
`MULTIPLE(n)`	the initial value of the semaphore counter is n ($n >= 0$).

SOURCE: defines a source station,

MULTIPLE(n): used alone defines a station of type SERVER.

The integer associated with the option MULTIPLE may be a constant or an expression. If the TYPE parameter is omitted the station is considered as a single server, provided that the SERVICE parameter appears in the description of the station. If the SERVICE parameter is also omitted, then no server is associated with the station and it is only a queue in which customers must be explicitly manipulated (see TRANSIT and MOVE procedures).

The specified TYPE parameter is assigned to all the stations identified by the NAME parameter. The expressions appearing in a TYPE parameter are evaluated at initiation time.

Example:

```
/DECLARE/ QUEUE UC,DK,CHANNEL;
          INTEGER NPROC;

/STATION/ NAME = UC;
          TYPE = MULTIPLE (NPROC);

/STATION/ NAME = DK;

/STATION/ NAME = CHANNEL;
          TYPE = RESOURCE;
```

The number of servers of the UC station is given by the integer NPROC. DK station is a simple server (default value); CHANNEL station is a nonshareable resource.

3.7.4 SCHED parameter

Syntax:

sched_parameter	→	`SCHED` = [*order*] [, [`FEFS`,] `EXCLUDE` *concurrency_sets*];
order	→	[`FIFO` | `LIFO` | `FILO`] [, `PRIOR`] [, `PREEMPT`] |
		[`RESEQUENCE` (queue [, class])] [, `QUANTUM` [(real)] | `PS`]
concurrency_sets	→	({class [, real]} [, ...]) [, ...] |
		(real [, ...])

Semantics:

The SCHED parameter describes the scheduling of the customers in the station. The parameters have the following meanings:

FIFO (first-in/first-out): customers are ordered according to their order of arrival ; FIFO is the default value of SCHED.

LIFO or FILO (last-in/first-out): customers are ordered according to the reverse order of their arrival.

PRIOR (priority): customers are ordered in the queue according to their relative priorities (CPRIOR attribute of customers). The highest priority customers come first. Equal priority customers are ordered FIFO (default) or LIFO, according to the specified option.

PREEMPT (preemption): a customer being served is preempted by an arriving customer having a higher priority level; the service of the preempted customer is resumed when this customer is reallocated a server.

EXCLUDE (mutual exclusion): this option applies to multiple server stations or infinite stations. Two customers belonging to the same *concurrency set* may not be served simultaneously. See below the definition of concurrency sets.

FEFS (first eligible/first served): this option should be used only with the EXCLUDE option.

FEFS specifies that when several customers are waiting in the queue of a multiple server station with concurrency sets, the first eligible customers seize the servers, *even if there are uneligible customers waiting before them.*

Without FEFS (default behaviour), the first customer in the queue seizes a server. The second one is served only if it is eligible. If not, then the server stays idle, *even if there are eligible customers behind.*

QUANTUM (quantum allocation): a server is allocated to customers for fixed length periods defined by the associated real number. The quantum value can be specified with the QUANTUM parameter of the /STATION/ command.

PS (processor sharing): all the customers are served simultaneously with a service rate divided by n, if n is the current number of customers in the station. It can be seen as the mathematical limit of the QUANTUM discipline when the quantum value tends to zero.

RESEQUENCE (resequencing): the customers are served in the very order that they departed from the specified reference queue (and possibly with the specified class).

The specified SCHED parameter is assigned to all the stations identified by the NAME parameter. The expressions appearing in a SCHED parameter are evaluated at initiation time.

Example:

```
/DECLARE/ QUEUE CPU, DK, PROC(5);
          CLASS X,Y;

/STATION/ NAME  = CPU;
          SCHED = LIFO, PREEMPT;
```

```
            /STATION/ NAME  = DK;

            /STATION/ NAME  = PROC(1), PROC(2);
                      SCHED = QUANTUM (0.2);
```

The CPU server has a LIFO preemptive service discipline, i.e. any arriving customer preempts the server. Station DK has a simple FIFO queue discipline (default value). Station PROC(1) and station PROC(2) are scheduled with a quantum algorithm, each quantum being 0.2 time units long.

Concurrency sets with classes

The concurrency sets are defined by means of customer classes and/or probabilities. Each concurrency set definition is enclosed between parentheses. The concurrency sets are defined with the following generic form:

$$\begin{array}{c}(c_1, p_{1,1}, c_2, p_{1,2}, \ldots c_n, p_{1,n}),\\ (c_1, p_{2,1}, c_2, p_{2,2}, \ldots c_n, p_{2,n}),\\ \vdots \\ (c_m, p_{m,1}, c_2, p_{m,2}, \ldots c_n, p_{m,n})\end{array}$$

which means that a customer of class c_j entering the station belongs to concurrency set i with the probability $p_{i,j}$. The default value of $p_{i,j}$ is 1. The sum of probabilities assigned to class c_j among concurrency sets must be less than or equal to 1. If class c_j is missing from a set description, the corresponding probability is zero.

Example:

```
      /DECLARE/ CLASS C1, C2, C3, C4, C5, C6;

      /STATION/ NAME = A;
                SCHED = EXCLUDE (C1, 0.4, C2, 0.5), (C3, 0.2, C4, C5);
```

A class C1 customer entering station A belongs to the first concurrency set with probability 0.4.

A class C4 customer entering station A always belongs to the second concurrency set.

Class C6 customers do not belong to any concurrency set.

Concurrency sets without classes

When there is only one customer class, the concurrency sets are defined with probabilities: $(p_1, p_2, \ldots p_m)$ A customer belongs to concurrency set j with probability p_j. The sum of probabilities must be less than or equal to 1.

Example:

```
      SCHED = (0.2, 0.4, 0.3);
```

A customer entering the station belongs to the first concurrency set with probability 0.2. It belongs to no concurrency set at all with probability $1 - (0.2 + 0.4 + 0.3)$.

3.7.5 PRIOR parameter

Syntax:

prior_parameter $\rightarrow$ `PRIOR` [(class_list)] = integer;

Semantics:

The PRIOR parameter determines the priority level of the customers entering the stations identified by the NAME parameter. This priority level may be different for each class of customers.

If a class list is specified, its expression is evaluated at compile time. All classes of the list will have the same priority.

The PRIOR parameter without class specification applies to all the classes for which no specific priority level is defined. Priority levels are positive integers (the greater the integer the higher the priority). The priority level may be a constant or an expression.

The expression defining the priority level in a PRIOR parameter is evaluated at initiation time (no dynamic evaluation during solution time).

If the PRIOR parameter is undefined for a given class, the customers belonging to this class keep their previous priority levels (zero is the default priority level for each newly created customer).

Note that affecting a priority level to a customer is not related to the fact that the station scheduling policy is PRIOR.

Example:

```
/DECLARE/ QUEUE UC,UK;
          INTEGER PX;
          CLASS X, Y, Z;

/STATION/ NAME       = UC;
          PRIOR (X,Y)= PX;
          PRIOR      = 2;

/STATION/ NAME       = UK;
          SCHED      = PRIOR, PREEMPT;
          PRIOR (Y)  = 1;
          PRIOR (Z)  = 3;
```

The customers entering the station UC get new priority levels: class X and class Y customers get a priority determined by variable PX (evaluated at solution time). Customers of the other classes get a priority level equal to 2. The customers treated in station UK are served in priority order (with possible preemption). Class Y customers and class Z customers get priority levels equal to 1 and 3 respectively. Class X customers keep their priority level (i.e. the one they had before entering the station UK).

3.7.6 QUANTUM parameter

Syntax:

quantum_parameter $\rightarrow$ `QUANTUM` [(class_list)] = real ;

Semantics:

The QUANTUM parameter specifies the service quantum allocation to each class in the stations identified by the NAME parameter. These values are used only if the QUANTUM option is specified in the parameter SCHED. There is no default value provided for a quantum

(SCHED = QUANTUM will produce an error at run time if no quantum value is specified). Each quantum value may be a constant or an expression.

The expression defining the quantum value in a QUANTUM parameter is evaluated at initiation time (no dynamic evaluation during solution time).

The QUANTUM parameter, without class option, is applied to all the classes for which no quantum value is specified. This default value may also be associated with the QUANTUM option of the SCHED parameter.

If a class list is specified, its expression is evaluated at compile time.

Example:

```
/DECLARE/ QUEUE UC;
          REAL QT;
          CLASS X, Y, Z;

/STATION/ NAME        = UC;
          SCHED       = QUANTUM (0.3);
          QUANTUM (X) = QT;

& equivalent description:

/STATION/ NAME        = UC;
          SCHED       = QUANTUM;
          QUANTUM (X) = QT;
          QUANTUM     = 0.3;
```

Service in station UC is allocated by quanta being 0.3 time units long for Y and Z classes and given by the variable QT for X class customers.

3.7.7 INIT parameter

Syntax:

init_parameter → `INIT` [(class_list)] = integer ;

Semantics:

The INIT parameter specifies the number of customers of each class being in the stations identified by the NAME parameter, at the beginning of solution time.

The INIT parameter, without the class option, is applied for all the classes for which no specific initial value exists. If a class list is specified, its expression is evaluated at compile time. The initial number of customers may be a constant or an expression.

The expression defining the initial number of customers in an INIT parameter is evaluated at initiation time.

The means by which customers are initially generated and placed in queues are described in Chapter 4.

Note that a resource or semaphore station may not contain customers at initialization time.

Example:

```
/DECLARE/ QUEUE A,B;
```

```
          CLASS X,Y;
          INTEGER N;

/STATION/ NAME     = A;
          INIT (X) = 2;
          INIT (Y) = 1;

/STATION/ NAME     = B;
          INIT     = N;
```

Station A is initialized with two customers of class X and one customer of class Y. Station B is initialized with N customers in both class X and class Y (N must have a value before calling a solver).

Note:
`INIT = N;` is a common trap for QNAP2 beginners. This parameter does not initialize the station with N customers but with N customers for *each class existing at initiation time.*

3.7.8 SERVICE parameter

Syntax:
service_parameter → `SERVICE` [(class_list)] = statement ;

Semantics:
The SERVICE parameter specifies the service required by the customers being in the stations identified by the NAME parameter.

The service may be different for each customer class. In this case the SERVICE parameter has to be indexed by the appropriate list of classes. A non-indexed service applies to all classes for which no service is specified.

If a class list is specified, its expression is evaluated at compile time. The SERVICE parameter is ignored in the case of a station defined as a resource or semaphore station.

A service is comprised of work demands and/or object manipulation operations.

The work demands are expressed in work units (e.g. number of instructions to be executed, number of bytes being transferred). These values are described by means of appropriate work demand procedures (e.g. CST, EXP). The service rate of one server of the considered station determines the effective service time. If this rate is equal to 1 (1 work unit per time unit) then the service time is equal to the work demand expressed in service units (1 is the default value of the RATE parameter).

The object manipulation operations may include synchronization procedures (P and V procedures on semaphores, TRANSIT or PRIOR procedures, etc.). These procedures control the activity of the servers of the station. The servers are blocked (i.e. have a null instantaneous service rate) if the synchronization conditions are not met.

The expressions and statements appearing in the SERVICE parameter are evaluated at solution time (see Chapter 5, section 5.1.2 “Bringing resolution into play,” for more details depending on the solver activated).

Permitted service specifications are generally limited by the solver used (the limits for each solver of QNAP2 are given in Chapter 5).

Example:

```
/DECLARE/ QUEUE A;
          CLASS X, Y, Z;
          REAL SA;

/STATION/ NAME        = A;
          SERVICE (X) = CST (4);
          SERVICE (Y) = HEXP (SA, 16);
          SERVICE     = EXP (5.67);
```

The service time of class X customers is defined by a constant work demand of 4 service units; the work demand of class Y customers is a hyper-exponential distribution having a mean equal to SA and a squared coefficient of variation equal to 16; class Z customers work demand is distributed according to an exponential distribution having a mean equal to 5.67 (default service for this station).

Example:

```
/DECLARE/ QUEUE A, SEM;

/STATION/ NAME    = A;
          SERVICE = BEGIN
                      EXP(12.);
                      P(SEM);
                      PRINT(TIME); CST(5.);
                      V(SEM);
                    END;
```

3.7.9 TRANSIT parameter

Syntax:

transition_parameter	→	`TRANSIT` [(class_list)] = transition_list ;
transition_list	→	{ queue_sublist [,class_sublist], real_sublist }[,...]
		queue_sublist [,class_sublist][, real_sublist]

Semantics:

The TRANSIT parameter describes the routing rules of customers at service completion time. The routing mechanism is defined as a probabilistic switch. The transition probabilities may be given for each destination station and for each class by giving a list of 3-uples. Each 3-uple consists of:

- one or several destination stations,
- one or several destination classes,
- one or several transition probabilities (relative or absolute).

Each element of a 3-uple may be a constant, an expression or a sublist. In the case of sublist elements, the sublists must have the same number of elements. However, the last element of the last sublist of transition probabilities may be omitted: its absolute value is then computed as the complement to 1 of the sum of the other probabilities. The second element of the 3-uple may be omitted if no class transitions are considered.

The elements of the 3-uples appearing in a TRANSIT parameter are evaluated at initiation time (no dynamic evaluation during solution time).

The optional class_list in the left-hand side references the classes from where, in the current queue, the transition is described. This list is evaluated at compile time. If no class is specified in the TRANSIT parameter, the values given apply for all the classes for which no explicit transition rule is specified.

In the case of an open queueing network, the predefined queue OUT is used to name the network exit (where customers are destroyed).

A SOURCE station creates customers of various classes that may be routed to stations and classes according to fixed probabilities.

Example:

```
& use of absolute probabilities

/DECLARE/ QUEUE A, B, C;

/STATION/ NAME    = A;
          SERVICE = EXP (5);
          TRANSIT = A, 0.5, B, 0.3, C;

& use of relative probabilities

          TRANSIT = A, 10, B, 6, C, 4;
```

At the end of their service time, customers go to station:

- A with the probability 0.5
- B with the probability 0.3
- C with the probability 0.2

Example:

```
/DECLARE/ QUEUE A,B,C;
          CLASS X, Y,Z;
          REAL PA, PB;

/STATION/ NAME        = A;
          SERVICE     = EXP (5);
          TRANSIT     = B;
          TRANSIT (X) = A,PA, B,Y,PB,C,Z;
          TRANSIT (Y) = A,Z,1,C,Y,2;
```

The TRANSIT parameter without class option (TRANSIT=B) applies as a default value for class Z for which no transition is defined. Customers move according to the following transition rules:

class	destination	probability
X	A(X)	PA
	B(Y)	PB
	C(Z)	1-PA-PB
Y	A(Z)	1/3
	C(Y)	2/3
Z	B(Z)	1

Example:

```
/DECLARE/ QUEUE Q1,Q2;
          CLASS X,Y;

/STATION/ NAME        = Q1;
          & ...
          TRANSIT(X,Y)= Q1,X,0.1,Q2,Y;
```

Class X and Y customers go to Q1 in class X with probability 0.1. They go to Q2 with probability 0.9.

Example:

```
& open network

/DECLARE/ QUEUE A;

/STATION/ NAME    = A;
          & ...
          TRANSIT = A, 0.2, OUT;
```

Customers leave the network (i.e. are destroyed) with a probability equal to 0.8 at service completion time in A.

Example:

```
& description of a source

/DECLARE/ QUEUE G, A, B;
          CLASS X,Y;

/STATION/ NAME    = G;
          TYPE    = SOURCE;
          SERVICE = CST (10);
          TRANSIT = A, X, 0.2, B, Y;
```

The source G generates customers at constant time intervals (10 time units). These customers are routed to station A with class X having a probability of 0.2 (i.e. there is a customer flow of 0.02 customers per time unit), and to station B with class Y having a probability of 0.8.

Example:

```
& usage of list
/DECLARE/ QUEUE CPU, DISK(10);

/STATION/ NAME   = CPU;
          TRANSIT= DISK, 1 REPEAT 10;
```

The transitions to each disk are made with equal probability.

Example:

```
/DECLARE/ QUEUE CPU,DISK(3);
          REAL PROB(3);

/STATION/ NAME    = CPU;
          TRANSIT = DISK(1 STEP 1 UNTIL 3),PROB, OUT;
```

Note that the corresponding sublists have the same number of elements. Transition to DISK(1) with probability PROB(1), transition to DISK(2) with probability PROB(2), transition to DISK(3) with probability PROB(3), transition to OUT with probability 1.–PROB(1)—PROB(2)—PROB(3).

Example:

```
/DECLARE/ INTEGER NC =10;
          QUEUE CPU, A;
          CLASS CX(NC);
          REAL PX(NC);

/STATION/ NAME    = CPU;
          TRANSIT = A REPEAT NC,CX,PX;
```

Since A is not an array, REPEAT NC is mandatory in order to assign the transitions to station A in the classes CX(i) with the probabilities PX(i).

3.7.10 SPLIT parameter

Syntax:
split_parameter $\rightarrow$ `SPLIT` [(class_list)] = { ({queue, class, count} [, ...]) [prob] } [, ...];

Semantics:
The SPLIT parameter must be used in place of the TRANSIT parameter to specify a customer split.

Different splits can be specified for each class. A split specification without class_list applies to all classes for which no split is defined. If a class_list is present, it is evaluated at compile time. The other expressions are evaluated at initiation time.

The split mechanism is defined as a probabilistic switch between several split possibilities. "prob" is a real expression giving the weight of each split possibility. If the sum of prob values is not 1, then all prob values are normalized.

The triples {queue, class, count} indicate the destination queue, the class of "pieces" and the number of "pieces" resulting from each split. A single customer may be split into any number of "pieces" which can be sent to different queues.

Example:

```
/DECLARE/ QUEUE SENDER, RECEIVER, NETW1, NETW2;
          CLASS MSG, SMALLPK, LARGEPK;

/STATION/ NAME = SENDER;
          SPLIT (MSG) = (NETW1, SMALLPK, 40,
                         NETW2, SMALLPK, 60)  0.8,
                        (NETW1, LARGEPK, 10)  0.2;
```

A customer of class MSG is split into 100 customers of class SMALLPK with probability 0.8. 40 of these new customers are sent to NETW1, and 60 to NETW2.

Note:
SPLIT/MATCH and FISSION/FUSION should not be confused. SPLIT is intended to work in conjunction with MATCH. FISSION and FUSION can be used independently. See section 4.10 for more explanations.

3.7.11 MATCH parameter

Syntax:

match_parameter	→	`MATCH` = { origin : join [: `PRIOR` (integer)] } [, ...] |
		`MATCH` = { origin : join [: `WEIGHT` (real)] } [, ...]
origin	→	(queue [, class_list])
join	→	({class, count} [, ...]) result_class

Semantics:
The MATCH parameter is used to specify a join up of customers representing "pieces" of a customer that was previously split with the SPLIT parameter. All the expressions are evaluated at initiation time.

origin references the split specification that created the "pieces".

join specifies the number of "pieces" of each class required to form the resulting customer, of class result_class. Partial joins are allowed, i.e., it is not mandatory to join *all* pieces simultaneously.

Several join specifications can be provided. The selection can be performed with integer priority levels (deterministic switch) or real weights (probabilistic switch).

Example:

```
/DECLARE/ QUEUE SENDER, RECEIVER, NETW;
          CLASS MSG, SMALLPK, LARGEPK;

/STATION/ NAME = RECEIVER;
          MATCH = (SENDER, MSG) : (SMALLPK, 100) MSG,
                  (SENDER, MSG) : (LARGEPK,  10) MSG;
```

100 customers of class SMALLPK or 10 customers of class LARGEPK are joined up into one customer of class MSG.

Note:

SPLIT/MATCH and FISSION/FUSION should not be confused. SPLIT is intended to work in conjunction with MATCH. FISSION and FUSION can be used independently. See section 4.10 for more explanations.

3.7.12 FISSION parameter

Syntax:

fission_parameter → `FISSION` [(class_list)] = { ({queue, class, count} [, ...]) [prob] } [, ...];

Semantics:

The FISSION parameter must be used in place of the TRANSIT parameter to specify a customer fission.

Different fission possibilities can be specified for each class. A fission specification without class_list applies to all classes for which no fission is defined. If a class_list is present, it is evaluated at compile time. The other expressions are evaluated at initiation time.

The fission mechanism is defined as a probabilistic switch between several fission possibilities. prob is a real expression giving the weight of each fission possibility. If the sum of prob values is not 1, then all prob values are normalized.

The triples {queue, class, count} indicate the destination queue, the class of "pieces" and the number of "pieces" resulting from each fission. A single customer may be fissioned into any number of "pieces".

Example:

```
/DECLARE/ QUEUE UNPACK;
          CLASS SCREWBOX, SCREW;

/STATION/ NAME = UNPACK;
          FISSION (SCREWBOX) = (SCREW, 100);
```

A single customer of class SCREWBOX is fissioned into 100 customers of class SCREW.

Note:

SPLIT/MATCH and FISSION/FUSION should not be confused. SPLIT is intended to work in conjunction with MATCH. FISSION and FUSION can be used independently. See section 4.10 for more explanations.

3.7.13 FUSION parameter

Syntax:

fusion_parameter → `FUSION` = { join [: `PRIOR` (integer)] } [, ...] |
`FUSION` = { join [: `WEIGHT` (real)] } [, ...]

join → ({class, count} [, ...]) result_class

Semantics:

The FUSION parameter is used to specify a fusion of customers into a single customer. All the expressions are evaluated at initiation time.

join specifies the number of customers of each class required to form the resulting customer, of class result_class.

Several join specifications can be provided. The selection can be performed with integer priority levels (deterministic switch) or real weights (probabilistic switch).

Example:

```
/DECLARE/ QUEUE SCREWIT;
          CLASS SCREW, BOLT, SCREWED;

/STATION/ NAME = SCREWIT;
          FUSION = (SCREW, 1, BOLT, 1) SCREWED;
```

One SCREW customer and one BOLT customer are fusioned into one SCREWED customer.

Note:

SPLIT/MATCH and FISSION/FUSION should not be confused. SPLIT is intended to work in conjunction with MATCH. FISSION and FUSION can be used independently. See section 4.10 for more explanations.

3.7.14 RATE parameter

Syntax:

rate_parameter $\rightarrow$ `RATE` [(class_list)] = real_list ;

Semantics:

The RATE parameter specifies how fast the server of a station (or each server in the multiple server station case) can handle the service requests of the customers present in the station.

This speed is measured in number of work units performed during each time unit (e.g. MIPS, KOPS).

The service rate may be constant or variable. If the service rate is constant, the service time of a customer is equal to its work demand divided by the service rate. If the service rate is variable (then it is the instantaneous service rate), its value varies according to the number of customers queueing (see the general form below).

The service rate may be different for each class of customers. If a class list is specified, its expression is evaluated at compile time. The RATE parameter without class specification is the nominal service rate of the server.

The parameter RATE, if associated with a class list, gives a multiplicative coefficient of the nominal rate for the customers of the classes in the list.

The general form of the RATE parameter is:

- `RATE` $= a_1, a_2, \ldots, a_n$; nominal rate (with $a_i = a_n$ for $i > n$)
- `RATE` $(x) = x_1, x_2, \ldots, x_n$; class x coefficients (with $x_j = x_n$ for $j > n$)

The instantaneous service rate for class x customers is given by the formula $a_i * x_j$ when the station contains exactly i customers and j class x customers.

A rate is specified by a constant or an expression. The default values are RATE = 1.0 and RATE (ALL CLASS) = 1.0.

The expressions defining the rates in a RATE parameter are evaluated at initiation time (no dynamic evaluation during solution time).

Example:
Constant rate

```
/DECLARE/ QUEUE A;
          CLASS X, Y, Z;
          REAL RX = 0.9;

/STATION/ NAME     = A;
          RATE     = 20.;
          RATE (X) = RX;
          SERVICE  = EXP (5);
```

The nominal service rate of station A server is 20. The mean service time of class Y and Z customers is therefore 0.25 time units. The service rate for class X customers is 20*0.9 (= 18). Therefore, the mean service time for class X customers is 5./18 (= 0.28) time units. The preceding description is also equivalent to:

```
/STATION/ NAME        = A;
          SERVICE     = EXP (0.25);
          SERVICE (X) = EXP (0.28);
```

Example:
Dependent service rate

```
/DECLARE/ QUEUE A;
          REAL TAB (4) = (23.1, 24.1, 35.4, 48.7);

/STATION/ NAME    = A;
          RATE    = TAB;
          SERVICE = EXP (1);
```

The instantaneous service rate depends on the number of customers being in the station. The mean work demand being one work unit, the service time of one customer equals the inverse of the service rate.

```
For 1 customer the service rate is    23.1,
    2 customers          "            24.1,
    3 "                  "            35.4,
    4 "                  "            48.7
(or more)
```

3.7.15 CAPACITY parameter

Syntax:
capacity_parameter → `CAPACITY` [(class_list)] = integer ;

Semantics:

The CAPACITY parameter specifies the maximum number of customers allowed in the station. The capacity may be different for each customer class. The default capacity is infinite.

If a class list is specified, its expression is evaluated at compile time. The expression defining the capacity is evaluated at initiation time.

If no class list is specified, the maximum capacity is a global capacity applying to all customer classes. If a class-specific capacity is specified as well as a global capacity, the smallest capacity prevails.

Example:

```
/STATION/ NAME = STORAGE;
          CAPACITY (large1, large2) = 7;
          CAPACITY (small) = 100;
          CAPACITY = 10;
```

The STORAGE station can hold at most 10 items, including no more than 7 "large1" and 7 "large2" items. The 100 capacity for "small" items cannot be reached in practice, as the global limit is 10.

Note:

The capacity limit includes customers being served as well as waiting customers.

3.7.16 REJECT parameter

Syntax:

reject_parameter → `REJECT` [(class_list)] = statement;

Semantics:

The REJECT parameter specifies the processing of customers that are rejected due to the limited capacity of their destination station.

The reject processing may be different for each customer class. If no class list is given, the reject processing applies to all classes for which no specific reject was specified.

If a class list is specified, its expression is evaluated at compile time.

The statement is evaluated at initiation time for analytical algorithms or at solution time for simulation. No default processing is defined, so this parameter is mandatory for finite capacity stations.

Example:

```
/STATION/ NAME = STORAGE1;
          CAPACITY = 20;
          REJECT (large1, large2) = BEGIN
            PRINT (TIME, "STORAGE capacity exceeded");
            TRANSIT (STORAGE2);
          END;
          REJECT (small) = SKIP;
          TRANSIT = PACK;
```

Assume STORAGE1 is full and a customer arrives. If its class is "large1" or "large2", a warning message is printed and the excess customer is transferred to station STORAGE2. It its class is "small", it just skips the station and directly goes to PACK.

Note:

1. With analytical solvers, the only allowed processing is a single call to the SKIP procedure, which causes the rejected customer to skip the station and proceed as if it had been served. In this case, the TRANSIT parameter must be used to specify the destination of the customer.
2. With the discrete event simulator, the reject statement can include any algorithmic language operation.
3. The reject statement is executed by the rejected customer. Specific customer attributes and functions are available to obtain information about the reject conditions.
4. The reject statement *must* ensure a proper fate to the rejected customer. It should include at least one of the following:
   - a work demand procedure, in order to wait for a free place in the station;
   - a *blocking* synchronization operation (e.g., P, WAIT);
   - a successful TRANSIT to another station or to the same station after making room.
5. If one of the above operations is performed successfully, then the reject condition is forgotten. The corresponding data are lost. A second reject becomes possible.
6. If the reject statement ends with a new reject condition and there was no delay or blocking synchronization, then the simulation is stopped with an error message.
7. Special case for RESOURCE and SEMAPHORE stations with limited capacity: if a customer attempts a P operation and the station is full, the P is *not* performed, and the customer executes the reject statement of the resource or semaphore.
8. In case of preemptive scheduling: a customer entering a station with preemptive scheduling may cause a lower priority customer to be rejected. The reject statement is executed by the rejected customer.

3.7.17 COPY parameter

Syntax:

copy_parameter $\longrightarrow$ `COPY` = queue ;

Semantics:

The COPY parameter defines one or several identical stations by copying the parameters of another station, at compile time.

The parameters SCHED, SERVICE, TRANSIT, INIT, RATE, PRIOR, TYPE and QUANTUM of the template station are copied into the corresponding parameters of the station(s) identified by the NAME parameter.

The queue list expression defining the target stations is evaluated at compile time.

Example:

```
/DECLARE/ QUEUE DK0,DK1,DK2,CPU;
          CLASS X,Y;
```

```
/STATION/ NAME      = DK0;
          TRANSIT   = CPU;
          SERVICE   = EXP (10);
          SCHED     = LIFO, PREEMPT;
          PRIOR (X) = 2;
          PRIOR (Y) = 1;

/STATION/ NAME      = DK1;
          COPY      = DK0;

/STATION/ NAME      = DK2;
          COPY      = DK0;
          PRIOR (Y) = 0;
```

The station DK1 is identical to station DK0, and station DK2 differs only by the PRIOR(Y) parameter.

Note:

If the station used as reference in a COPY parameter is altered during the model description, these modifications are not reflected in the copied stations.

3.8 /CONTROL/ Command

3.8.1 Overview

Syntax:

```
control_command    →  /CONTROL/ [ control_parameter ][...]
control_parameter  →  option_parameter           |
                      unit_parameter             |
                      nmax_parameter             |
                      marginal_parameter         |
                      class_parameter            |
                      accuracy_parameter         |
                      estim_parameter            |
                      alias_parameter            |
                      tstart_parameter           |
                      tmax_parameter             |
                      period_parameter           |
                      test_parameter             |
                      random_parameter           |
                      trace_parameter            |
                      entry_parameter            |
                      exit_parameter             |
                      correlation_parameter      |
                      convergence_parameter      |
                      statistics_parameter
```

Semantics:

The /CONTROL/ command alters the internal control parameters of QNAP2 (compilation and solution phases). This command must only be used to change the default values of these control parameters.

The control parameters are grouped as follows:

- **input-output controls:**

`OPTION`	:	printing control,
`UNIT`	:	input-output units assignment,

- **resolution controls:**

`CLASS`	:	statistics for each customer class,
`MARGINAL`	:	marginal probabilities,
`ENTRY`	:	initialization sequence,
`EXIT`	:	exit sequence,
`CONVERGENCE`	:	numerical convergence control,

- **simulation controls:**

TSTART	:	starting time of measurements during a simulation run,
TMAX	:	maximum duration of the simulation,
PERIOD	:	test period,
RANDOM	:	random number generator seed,
TRACE	:	event trace,
ACCURACY	:	list of queues and classes with confidence intervals,
ESTIMATION	:	selection of the confidence intervals estimation method,
STATISTICS	:	selection of partial or global results
TEST	:	test sequence,
CORRELATION	:	test of measurements independence,

- **miscellaneous:**

NMAX	:	maximum number of classes,
ALIAS	:	definition of alias names.

3.8.2 OPTION parameter

Syntax:

```
option_parameter  →  OPTION = option [,...]
option            →  SOURCE        |
                     NSOURCE       |
                     RESULT        |
                     NRESULT       |
                     TRACE         |
                     NTRACE        |
                     DEBUG         |
                     NDEBUG        |
                     VERIF         |
                     NVERIF        |
                     WARN          |
                     NWARN         |
                     SIMPAR        |
                     NSIMPAR
```

Semantics:

The options have the following meanings (the default options are underlined).

`SOURCE`, `NSOURCE`: printing (or not) of the QNAP2 program lines following this control statement.

`RESULT`, `NRESULT`: printing (or not) of the final results of the analysis (in all cases these results are available by means of the result access functions and may be printed using the OUTPUT procedure).

`TRACE`, `NTRACE`: in the case of mathematical methods, printing of the intermediate results; in the case of the simulation, printing of the complete event trace.

`DEBUG`, `NDEBUG`: when the model is solved with the discrete event simulator, DEBUG enables use of the interactive source level debugger. This option should not be used with the analytical solvers.

`VERIF`, `NVERIF`: enables some tests during execution (e.g. array subscripts testing, references coherence). NVERIF may be used with tested models to reduce execution times.

`WARN`, `NWARN`: printing (or not) of the QNAP2 warning messages.

`SIMPAR`, `NSIMPAR`: activation (or not) of the parallelization of replications (if it is allowed).

Example:

In the following example several analyses are performed without any immediate result printing.

```
/CONTROL/ OPTION = NSOURCE, NRESULT;
/DECLARE/ QUEUE CPU;
          INTEGER N;

/STATION/ NAME = CPU;
          INIT = N;

/EXEC/    BEGIN
            N:=0;
            WHILE MBUSYPCT (CPU) < 0.8
            DO BEGIN
                 N:= N+1;
                 SOLVE ;
               END;
          END;

/EXEC/    OUTPUT;
```

As long as the CPU utilization rate remains lower than 80%, the number of customers is incremented and no result is printed. The results are printed only for the last solution performed (OUTPUT procedure).

3.8.3 UNIT parameter

Syntax:

```
unit_parameter  →  unit (file) [,...];
unit            →  OUTPUT          |
                   PRINT           |
                   INPUT           |
                   GET             |
                   LIBR            |
                   TRACE
```

Semantics:

The UNIT parameter controls QNAP2 input-output files.

OUTPUT: corresponds to the printing of the model source program (if OPTION=SOURCE) and to the printing of the results (if OPTION=RESULT) as well as error and warning messages,

PRINT: corresponds to the output produced by the PRINT procedure, as well as the default output of the WRITE and WRITELN procedures,

INPUT: corresponds to the file containing the source program to be analyzed,

GET: corresponds to the default input file for the GET and GETLN functions,

LIBR: corresponds to the default library file for the SAVE and RESTORE procedures,

TRACE: corresponds to the standard trace output file.

The default values of these parameters are the predeclared QNAP2 input-output files: FSYSOUTPUT, FSYSINPUT, FSYSPRINT, FSYSGET, FSYSLIB, FSYSTRACE. These files are normally assigned properly when QNAP2 is launched. The UNIT parameter is provided to redirect the standard input-output when needed.

Example:

```
/DECLARE/ INTEGER I, L(10);
          REAL R;
          FILE F;

/EXEC/ BEGIN
         FILASSIGN (F, "model.dat");
         OPEN (F, 1);                & read mode
       END;

/CONTROL/ UNIT = GET (F);

/EXEC/    BEGIN
            FOR I:= 1 STEP 1 UNTIL 7
                DO L (I):= GET (INTEGER);
            R:= GET (REAL);
          END;
```

3.8.4 CLASS parameter

Syntax:
class_parameter → CLASS = queue_list ;

Note:
This parameter is provided for compatibility with previous QNAP2 releases and might be removed in the future. When using the discrete event simulator, the SETSTAT:CLASS procedure should be used to specify the required class results. Refer to section 5.5 "Simulation results" for details.

Semantics:
The CLASS parameter specifies for which stations detailed results must be displayed. For these stations the occupation rate, mean service time, mean response time, mean customer number and the throughput are given for each customer class.

These results are printed in the final report of a solution and may be used in a model by means of the result access functions with a station name and a class name as parameters.
The queue list expression is evaluated at initiation time.

- CLASS = ALL QUEUE means that the results are requested for all the stations of the network (objects of type QUEUE).
- CLASS = NIL means that no results per class are requested (default value).

Example:

```
/DECLARE/ QUEUE A,B,C;
          CLASS X, Y;

/EXEC/    SOLVE; & no class result is produced

/CONTROL/ CLASS = A, B;
          & class results are produced
          & for stations A and B
/EXEC/    SOLVE;

/CONTROL/ CLASS = A;
          & class results are produced
          & for station A only
/EXEC/    SOLVE;

/CONTROL/ CLASS = ALL QUEUE;
          & class results are produced
          & for stations A,B and C
/EXEC/    SOLVE;

/CONTROL/ CLASS = NIL;
          & class results are not needed anymore
/EXEC/    SOLVE;

/EXEC/    BEGIN
            PRINT ( MRESPONSE (A, X) - MSERVICE (A, X));
            PRINT (MTHRUPUT (B , Y));
          END;
```

3.8.5 MARGINAL parameter

Syntax:
marginal_parameter → `MARGINAL` = { queue_sublist [, integer]} [,. . .];

Note:
This parameter is provided for compatibility with previous Qnap2 releases and might be removed in the future. When using the discrete event simulator, the SETSTAT:MARGINAL

procedure should be used to specify the required marginal probabilities. Refer to section 5.5 "Simulation results" for details.

Semantics:

The MARGINAL parameter specifies the stations for which the marginal probabilities are needed. The marginal probabilities are the probabilities that these stations contain exactly 0, 1, 2, 3,..., n customers respectively, globally (for all the customer classes) or for each class separately.

The marginal probabilities per customer class are computed only if the results per class are also requested for the station (see CLASS parameter).

The optional integer indicates the order up to which these probabilities have to be computed. This integer corresponds only to the previous queue_sublist. The default option is to compute the marginal probabilities up to the fifth order.

These probabilities and the variance of the customer number are printed together with the general results printed for the stations (if OPTION = RESULT). In addition, they may be used in the QNAP2 program using the function PCUSTNB and VCUSTNB.

The queue sublist and integer expressions are evaluated at initiation time.

- MARGINAL = ALL QUEUE means that the marginal probabilities have to be computed for all the stations of the model.
- MARGINAL = NIL means that no marginal probability is to be computed (default value).

Example:

```
/DECLARE/ QUEUE A, B, C;
          CLASS X, Y;

/CONTROL/ MARGINAL = ALL QUEUE;
/EXEC/    SOLVE;

& the marginal probabilities
& are computed for classes A, B
& and C up to the fifth order:

&     PCUSTNB(0,A) PCUSTNB(1,A) ... PCUSTNB(5,A)
&     PCUSTNB(0,B) PCUSTNB(1,B) ... PCUSTNB(5,B)
&     PCUSTNB(0,C) PCUSTNB(1,C) ... PCUSTNB(5,C)

/CONTROL/ CLASS = ALL QUEUE;
/EXEC/    SOLVE;

& the marginal probabilities are
& now computed for each class:

&     PCUSTNB (0,A,X) ... PCUSTNB (5,A,X)
&     PCUSTNB (0,A,Y) ... PCUSTNB (5,A,Y)
&     PCUSTNB (0,B,X) ... PCUSTNB (5,B,X)

/CONTROL/ MARGINAL = B, A, 20;
          CLASS = NIL;
```

```
/EXEC/    SOLVE;

& the global marginal probabilities
& are computed only for station A (up to
& the twentieth order) and for station B
& up to the fifth.

/CONTROL/ MARGINAL = ALL QUEUE, 10;
/EXEC/    SOLVE;

& the global marginal probabilities are
& computed up to the tenth order and for
& all the stations of the network.
```

3.8.6 CONVERGENCE parameter

Syntax:

convergence_parameter $\longrightarrow$ CONVERGENCE = real_list ;

Semantics:

The CONVERGENCE parameter is used to modify the default values of some parameters of the mathematical solvers (MARKOV, SOLVE). The precise meaning of these values is described in Chapter 5.

parameter 1: maximum number of iterations (default value 100),

parameter 2: precision of the termination test (default value 1E-6),

parameter 3: number of test vectors used (default value 10), in the Markovian solver,

parameter 4: ratio between the number of non-null entries in the state transition matrix and the number of states (default value 5), in the Markovian solver.

The default value is CONVERGENCE = 100, 1E-6, 10, 5.

The real list expression is evaluated at initiation time.

Example:

```
/CONTROL/ CONVERGENCE = 500,1E-6,5,5;

/EXEC/    MARKOV;

& Resolution with 500 iterations, 5 test
& vectors and a precision of 1.E-6

/CONTROL/ CONVERGENCE = 1000;
/EXEC/    MARKOV;

& up to 1000 iterations may be used
& others parameters keep their default values
```

3.8.7 TSTART parameter

Syntax:

tstart_parameter $\rightarrow$ `TSTART` =real ;

Semantics:

The TSTART parameter defines the time when the measurements on the simulated model are started in order to produce statistical estimations of the steady-state performance criteria. It can be used to reduce the bias due to the transient behaviour of the simulated model. The value of TSTART may be a constant or a real expression.

The default value is TSTART = 0.

The real expression is evaluated at initiation time.

3.8.8 TMAX parameter

Syntax:

tmax_parameter $\rightarrow$ `TMAX` =real ;

Semantics:

The TMAX parameter defines the maximum duration of a simulation run, expressed in units of time of the model itself. The duration may be defined by a constant or an expression.

The default value is TMAX = 0.

The real expression is evaluated at initiation time.

Example:

```
/DECLARE/ REAL TIMEMAX;

/CONTROL/ TMAX = TIMEMAX;

/EXEC/    FOR TIMEMAX:= 1000.,5000.,10000.
              DO SIMUL;
```

Note:

The maximum simulation time may also be set with the SETTMAX procedure. SETTMAX may also be used during the simulation. See the Reference Manual for details.

3.8.9 RANDOM parameter

Syntax:

random_parameter $\rightarrow$ `RANDOM` = integer ;

Semantics:

The random numbers generated by the random number generation functions (see the QNAP2 Reference Manual for the list of these functions) are produced by a single uniform pseudo-random number generator.

This generator is reinitialized at the beginning of each execution time phase (/EXEC/ command). The reinitialization value may be changed by the parameter RANDOM. The default value of RANDOM is 413.

The value of RANDOM may be an integer constant or an integer expression. The integer expression is evaluated at execution time.

Note that the generator is not reinitialized at initiation time. As a consequence, if several simulation runs are launched within one /EXEC/ command, a single random stream is used throuhout the different runs so that the random stream for the second run starts where the stream for the first run ended, and so on.

Example:

```
/DECLARE/ QUEUE A, B;
           INTEGER I;

/CONTROL/ TMAX=10;

& 3 simulations are run with independent
& random streams
/EXEC/    FOR I:= 1 STEP 1 UNTIL 3 DO SIMUL;

& 2 simulations are run with identical
& random streams
/EXEC/ SIMUL;
/EXEC/ SIMUL;

& 1 simulation run with a new random
& stream
/CONTROL/ RANDOM = 337;
/EXEC/ SIMUL;
```

3.8.10 ACCURACY parameter

Syntax:

accuracy_parameter → `ACCURACY` = { queue_sublist [,class_sublist] } [,...];

Note:

This parameter is provided for compatibility with previous QNAP2 releases and might be removed in the future. When using the discrete event simulator, the SETSTAT:ACCURACY procedure should be used to specify the required confidence intervals. Refer to section 5.5 “Simulation results” for details.

Semantics:

The ACCURACY parameter specifies the stations and classes for which confidence intervals are computed during a simulation run. The confidence intervals are used to assess the accuracy of the results produced by the simulation run (as compared to the theoretical stationary solution). The confidence intervals are produced for all the standard performance criteria (service, response and blocing time, number of customers, utilization rate) of the elements (queue, class) defined by the sublist expressions (queue_sublist [,class_sublist]).

The sublist expressions are evaluated at initiation time.

- ACCURACY = ALL QUEUE, ALL CLASS means that the confidence intervals are required for all the stations and all the classes of the network.
- ACCURACY = NIL means that no confidence interval is required (default value).

If a class_sublist is associated with a queue_sublist, the queues listed in the queue_sublist must appear also in the parameter CLASS.

The confidence intervals produced can be accessed by the confidence intervals access functions. These functions are similar to the results access functions and are available for the same performance criteria. The complete list of these functions is given in the QNAP2 Reference Manual.

The selection of the confidence intervals estimation method used is controlled by the ESTIMATION parameter. This parameter has to be specified to obtain the confidence intervals calculation.

3.8.11 ESTIMATION parameter

Syntax:

```
estim_parameter  →  ESTIMATION = method ;
method           →  REPLICATION (integer)   |
                    REGENERATION            |
                    SPECTRAL [(real)]
```

Semantics:

The ESTIMATION parameter controls the method used to estimate the confidence intervals requested. Three methods for confidence intervals estimations are included in QNAP2. These methods are presented in the section 5.4 "Simulation" of the Chapter 5 of this manual. Only a brief overview is given here.

REPLICATION: The confidence intervals are computed using the replication method . The integer parameter is mandatory in order to specify the number of replications requested. The maximum duration of each replication is defined by the parameter TMAX. The execution of the STOP procedure in the current replication terminates this replication and starts the next one.

REGENERATION: The confidence intervals are computed using the regeneration method . Exact or approximate regeneration points must be defined by explicit calls to the SAMPLE procedure in a service description or a test sequence.

SPECTRAL: The confidence intervals are computed using the spectral method . The real parameter is optional: it is used to defined the first measurement period of the method. The default value of this parameter is TMAX/512.

3.8.12 STATISTICS parameter

Syntax:

```
statistics_parameter  →  STATISTICS = option ;
option                →  PARTIAL | GLOBAL
```

Note:

This parameter is provided for compatibility with previous QNAP2 releases and might be removed in the future. When using the discrete event simulator, the SETSTAT:PARTIAL procedure should be used to request partial statistics. Refer to section 5.5 "Simulation results" for details.

Semantics:

When the regeneration method is used (ESTIMATION = REGEN), the STATISTICS parameter controls the type of results (partial or global) which are available at the end of each regeneration period.

PARTIAL: During a simulation run, a call to the procedure OUTPUT or to a result access function will produce the statiscal results collected during the last regeneration period (i.e. during the interval of time between the two last calls to the procedure SAMPLE). This facility applies only to the following performance criteria (globally or for each class): mean service time, busy percentage, mean number of customer, mean response time, mean blocked time, throughput.

After the end of a simulation run, a call to the procedure OUTPUT or to a result access function will produce the statistical results collected during the complete simulation run (i.e. during the interval of time between the measurement starting time and the end of the simulation).

GLOBAL: During a simulation run, a call to the procedure OUTPUT or to a result access function will produce the statistical results collected during the interval of time between the measurement starting time and the last regeneration point (i.e. last call to the procedure sample).

After the end of a simulation run, a call to the procedure OUTPUT or to a result access function will produce the statistical results collected during the complete simulation run (i.e. during the interval of time between the measurement starting time and the end of the simulation).

3.8.13 CORRELATION parameter

Syntax:

correlation_parameter → `CORRELATION` = { queue_sublist [, integer] [,class_sublist] } [,. . .] ;

Note:

This parameter is provided for compatibility with previous QNAP2 releases and might be removed in the future. When using the discrete event simulator, the SETSTAT:CORRELATION procedure should be used to specify the required aut-correlation coefficients. Refer to section 5.5 "Simulation results" for details.

Semantics:

When the regeneration method is used, the CORRELATION parameter controls the computation of the auto-correlation functions of the measurements made during each regeneration intervals of the simulation run. These functions help in determining the validity of the independance assumptions made on these measurements (this assumption is used for the calculation of the confidence intervals with the regeneration method). If this assumption is not verified the produced confidence intervals may be meaningless.

- CORRELATION = ALL QUEUE, ALL CLASS means that the functions have to be computed for all the stations and all the classes of the network.
- CORRELATION = NIL means that no correlation is required (default value).

The computation of the correlation function is performed for the stations and classes specified by the queue sublist and class sublist expressions. The elements (queue, class) appearing in the parameter CORRELATION must also be specified in the parameter ACCURACY. The maximum order of the auto-correlation function may be specified by the integer expression (the default value for this order is 5).

The sublist and integer expressions are evaluated at initiation time. The CORRELATION parameter may only be used with the regeneration method (ESTIMATION = REGEN).

3.8.14 PERIOD parameter

Syntax:

period_parameter → `PERIOD` = real ;

Semantics:

The PERIOD parameter determines the activation period of the test sequence (see TEST parameter) This period may be determined by a constant or by an expression which is evaluated at initiation time.

- If PERIOD = 0. is specified then the test sequence is activated at each instant corresponding to an event during the simulation run. Therefore, it is possible to permanently test the model state (active wait) and to perform some operations if the model is in some determined states. It should be noted that this mechanism is costly and therefore should only be used if expressly required.
- There is no default value for the parameter PERIOD. If the parameter has not been defined the test sequence, if any, is not activated.
- PERIOD = ; cancels any previous definition of PERIOD.

3.8.15 TEST parameter

Syntax:

test_parameter → `TEST` = statement ;

Semantics:

The TEST parameter is used to specify a statement to be executed, during a simulation run, at the end of each interval determined by the PERIOD parameter.
In a test sequence the user may, for instance:

- define regeneration points using the SAMPLE procedure in order to split the simulation duration into independent fixed or variable length intervals as required by the regeneration method.
- print the results obtained for each station during the current period by using for example the OUTPUT procedure and therefore keep track of the station behaviour during the simulation.

- stop the simulation run before the completion of the simulation duration TMAX by using the STOP procedure.

All these actions may be conditionned by tests on the current state of the model (current queue lengths, number of services completed,. . .).
The default value is `TEST = ;`.

The statement associated with a TEST parameter is executed during solution time.

Example:

```
/DECLARE/ QUEUE A,B,C;

& the OUTPUT procedure is called 10
& times (every 200 time units).

/CONTROL/ PERIOD = 200.;
          TMAX = 2000.;
          TEST = OUTPUT;

/EXEC/    SIMUL;

& the simulation run will stop after
& 2000 time units or if more than 500
& services have been completed.
& TEST is activated every 10 time units.

/CONTROL/ PERIOD = 10.;
          TEST = IF A.NBOUT > 500 THEN STOP;

/EXEC/    SIMUL;

& after each event of the model one tests if
& stations A,B and C are both empty and if this
& condition is true then the SAMPLE procedure
& is called in order to create a regeneration point.

/CONTROL/ PERIOD = 0.;
          ESTIMATION = REGENERATION;
          TEST=IF A.NB+B.NB+C.NB=0 THEN SAMPLE;

/EXEC/    SIMUL;
```

3.8.16 TRACE parameter

Syntax:

trace_parameter → TRACE = real [,real] [,string];

Note:

This parameter is provided for compatibility with previous QNAP2 releases and might be removed in the future. When using the discrete event simulator, the SETTRACE:*keyword* procedures should be used to specify the required tracing options. Refer to chapter 9 for details.

Semantics:

The TRACE parameters controls starting or ending time of the event trace during a simulation run. The first value is the starting time and the second one is the ending time (default value: TMAX).

These values may be specified by a constant or an expression which is evaluated at initiation time.

It should be noted that for a simulation run, the option TRACE of the OPTION parameter corresponds to TRACE= 0.,TMAX (i.e. trace during the whole simulation run). The default value is TRACE = 0.0, 0.0 which means that no trace is produced.

The string argument permits to cpecify "L80" (default value) for a trace on 80 columns or "L132" for an output on 132 columns.

Example:

```
& the trace is active during the whole simulation
/CONTROL/ TMAX = 2000.;
          TRACE= 0., 2000., "L132";
/EXEC/    SIMUL;

& the trace is active from the beginning
& and until time 500 with 132 columns

/CONTROL/ TRACE= 0.,500.;
/EXEC/    SIMUL;

& the trace starts at time 500 up to
& the end of the simulation run.

/CONTROL/ TRACE = 500.;
/EXEC/    SIMUL;

& the trace is active during the whole simulation

/CONTROL/ OPTION = TRACE;
/EXEC/    SIMUL;
```

3.8.17 NMAX parameter

Syntax:

nmax_parameter $\rightarrow$ `NMAX` = integer;

Semantics:

The NMAX parameter is used to increase the maximum number of classes available in a model. This parameter must be specified before the first declaration of a queue or of a class and may not be subsequently changed.

The default value is NMAX = 20. The value of the NMAX parameter may be specified as a constant or an expression which is evaluated at compile time.

3.8.18 ENTRY parameter

Syntax:

entry_parameter → ENTRY = statement ;

Semantics:

The ENTRY parameter describes an instruction sequence that must be executed before each analysis of the model (just before the solution starts). In this instruction sequence, the user may initialize variables used in station descriptions or perform various printing.

ENTRY=; cancels previous definitions (default value).

The statement associated with an ENTRY parameter is executed at the beginning of solution time (in the case of a simulation run with the replication method, the ENTRY statement is executed once only at the beginning of the first replication).

Example:

```
/DECLARE/ QUEUE A,B,C;
          INTEGER MA, MB, MC;
          REAL PA, PB, PC;

/STATION/ NAME = A;
          TRANSIT = B, PB, C, PC, A;
          & ...

/CONTROL/ ENTRY = BEGIN
                    PRINT(" MA, MB, MC ",MA, MB, MC);
                    PB:= MB; PB:= PB / (MA+MB+MC);
                    PC:= MC; PC:= PC / (MA+MB+MC);
                  END;

/EXEC/    BEGIN
            MA:= 10;
            MB:= 20;
            MC:= 30;
            SOLVE ;
            MA:= 20;
            SOLVE ;
          END;

/EXEC/    FOR MC:= 50, 100 DO SOLVE ;
```

The statements associated with the ENTRY parameter are executed at each call to the SOLVE procedure. In this sequence the transition probabilities PB and PC are computed as functions of variables MA, MB, and MC.

3.8.19 EXIT parameter

Syntax:

exit_parameter → `EXIT` = statement ;

Semantics:

The EXIT parameter describes an instruction sequence that is to be executed after each solution of the model (when returning from the solution procedure). In this sequence, the user may compute additional results or perform some printing.
EXIT=; cancels previous definitions (default value).

The statement associated with an EXIT parameter is executed at the end of solution time (in the case of a simulation run with the replication method, the EXIT statement is executed once only at the end of the last replication).

Example:

```
/DECLARE/ QUEUE A, B, C;
          REAL LRESPONS (20);
          INTEGER N;

/CONTROL/ EXIT = BEGIN
                   LRESPONS (N):= MRESPONS (C);
                   PRINT (MRESPONS (A)+ MRESPONS (B));
                   PRINT ("END OF ANALYZIS");
                 END;

          OPTION = NRESULT;

/EXEC/    FOR N:= 1 STEP 1 UNTIL 20 DO SOLVE ;
```

3.8.20 ALIAS parameter

Syntax:

alias_parameter → `ALIAS` = {(identifier,identifier)} [,...];

Semantics:

The ALIAS parameter is used to add new names to existing identifiers or keywords (command or parameter names), except aliases themselves. The first identifier is the alias given to the second identifier.

Current implementation restricts key words to be non-reserved ones (e.g. BEGIN, END, ALL, WITH,...). Several aliases may be given to a given identifier.

Example:

French aliasing.

```
/DECLARE/ QUEUE A, CPU;

/CONTROL/ ALIAS=(NOM, NAME),
                                    (DEBIT, THRUPUT),
                                    (GESTION, SCHED);

                              ALIAS=(PAPS, FIFO);

               /STATION/ NOM=CPU;
                              GESTION=PAPS;
```

In this example, /CONTROL/ ALIAS = (N, NOM); is not permitted since NOM is an alias of NAME.

Modeling Mechanisms $\boxed{4}$

4.1 Introduction

This chapter describes the main mechanisms used in a queueing network model (queue disciplines, service disciplines, synchronization procedures, ...).

Some features are meaningful only when some particular solver is to be used (e.g. SIMUL, MARKOV, ...). Although some of these features may be redundant or even illegal at run time when used with an improper solver, they imply no a-priori restriction on the resolution capabilities of QNAP2 since the SIMUL procedure will work on any legal QNAP2 model.

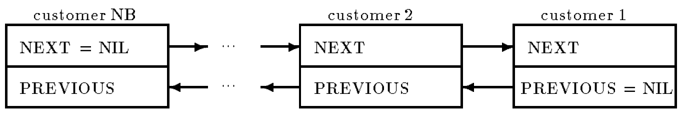
4.2 Queue Organization

4.2.1 Queue structure

Customers belonging to the same queue are chained by pointers: a pointer to the previous customer (towards the beginning of the queue: PREVIOUS attribute), and a pointer to the next customer (towards the end of the queue: NEXT attribute).

The first customer in a queue is pointed to by the FIRST attribute of the queue. For instance, when a SINGLE SERVER queue is busy, FIRST points to the customer which is being served. The last customer in the queue is pointed to by the LAST attribute of the queue.

In an empty queue: FIRST = LAST = NIL. Both the pointer to the customer preceding the first customer and the pointer to the customer following the last one are empty: FIRST.PREVIOUS = LAST.NEXT = NIL.



Example:

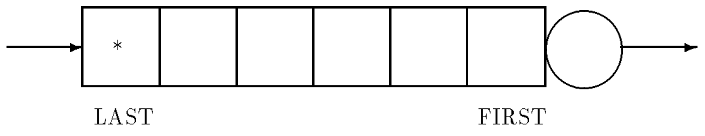
The following example searches in a queue for the customer having the highest value of attribute X (X >= 0). The search begins with the FIRST customer.

```
/DECLARE/ QUEUE A;
          CUSTOMER INTEGER X;
          REF CUSTOMER C, CMAX;
          INTEGER XMAX;

/EXEC/    BEGIN
            MAX:= 0; C:= A.FIRST;
            WHILE C <> NIL DO
                  BEGIN
                    IF C.X > XMAX THEN
                       BEGIN
                         CMAX:= C; XMAX:= C.X;
                       END;
                    C:= C.NEXT;
                  END;
            PRINT (" MAXIMUM = ", XMAX);
          END;
```

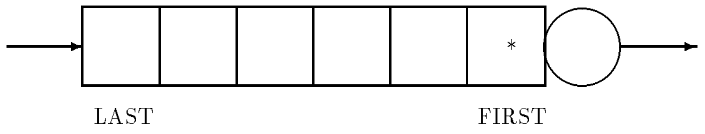
4.2.2 Queue discipline

The general rule is that a customer arriving at a station is assigned a position inside the queue according to the SCHED parameter of the station. This rule does not apply when the placement of the customer is forced by the procedures BEFCUST or AFTCUST (see section 4.9 Customer Forced Transition).

FIFO: The incoming customer is placed at the end of the queue (thus becoming the LAST customer in the queue).

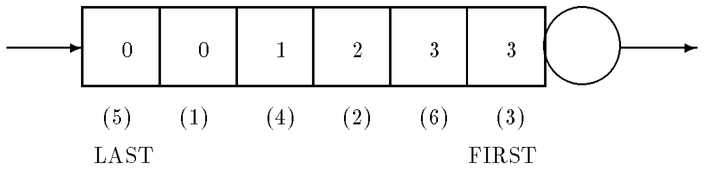


LIFO: The incoming customer is placed at the head of the queue (thus the FIRST attribute of the queue always references the last arrived customer).



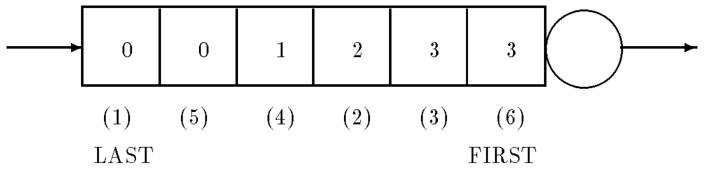
FIFO, PRIOR: Each customer has a priority level (CPRIOR attribute of the customer). This attribute may be modified during a transition by means of the TRANSIT procedure, the BEFCUST procedure or the 'AFTCUST procedure', directly by the PRIOR procedure or by the PRIOR parameter of the station.

The customers are ranked by priority level: those with the highest priority level being at the head of the queue. Customers with equal priority levels are ordered according to a FIFO discipline.



The numbers in the boxes are the priority levels. The parenthesized numbers show the order of arrival in the queue.

LIFO, PRIOR: The customers are ordered according to their priority levels, the customers with the highest priority levels being at the head of the queue. Customers with equal priority levels are ordered according to a LIFO discipline.



RESEQUENCE: The customers are ordered in the exact order that they departed from the reference queue.

The order in which they arrive at the resequencing queue has no importance. A customer arriving before its preceeding customers simply waits in the queue until all its preceeding customers arrived. A customer arriving after its succeeding customers seizes the server.

4.2.3 Queue capacity

The default queue capacity is infinite.[1]

Limited capacity queues can be specified with the CAPACITY parameter of the /STATION/ command. This does not change the basic queue organization.

The CAPACITY attribute of QUEUE objects returns the capacity limit, or -1 if the capacity is not limited. `queue.CAPACITY` returns the global limit; `queue.CAPACITY (class)` returns the limit for the specified class.

Note:

As the queue capacities are evaluated during initiation time, this attribute should not be used until solution time.

Example:

```
/STATION/ NAME = STORAGE;
          CAPACITY = 20;
          SERVICE = BEGIN
            PRINT (STORAGE.CAPACITY);    & yields 20
            ...
          END;

/EXEC/ PRINT (STORAGE.CAPACITY);         & undefined result
```

Customers which cannot fit into a limited capacity station are rejected. A customer can be rejected due to several reasons:

- It was normally completing a service in another station, and attempted to transit to a limited capacity station.
- It was forced to transit to a limited capacity station by another customer.
- It was the lowest priority customer in a limited capacity station, and a higher priority customer came in.
- It requested a pass grant (P procedure) from a semaphore or resource station with limited capacity.
- It was forced by another customer to request a pass grant from a semaphore or resource station with limited capacity.
- It was rejected, waited some time and tried to transit to a limited capacity station.
- It was rejected, waited some time and tried request a pass grant from a limited capacity resource or semaphore.

Rejected customers execute the statement specified with the REJECT parameter of the /STATION/ command. The following customer attributes are available in order to obtain information about the reject conditions:

QREJECT is a reference to the queue from which the customer is rejected.

CLREJECT is a reference to the class which caused the reject, when it was due to a class-specific capacity limit.

CRJCAUTH is a reference to the higher priority customer which caused the reject, when it was due to preemption.

[1]Actually, it is limited by the available memory, which is installation dependent.

CQUEUE is a reference to the queue that the customer is leaving. Its value is NIL if the customer was just created with the NEW function, or if it was rejected due to preemption.

Note:

Those attributes are available within the reject statement only, and *before* any delay or synchronization operation is performed by the rejected customer.

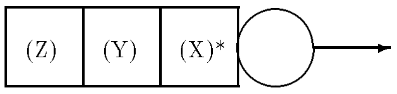
4.2.4 Queue initialization

If the initial state of a queue involves several customer classes, the initial customers are ordered according to the following rules:

- the number of customers of each class is determined by the INIT parameter of the station;
- the customers are generated and sent to the station according to the declaration order of the classes (customers of the first declared class are created first and so on);
- the customers are ordered according to the SCHED parameter of the station but the server allocation is not yet started; if the station has a priority discipline the customers of the highest priority level will be at the head of the queue;
- when all the customers defined by the INIT parameter are in the queue the server allocation is started. Therefore, in a station with a priority discipline, the customers with the highest priority levels are served first.

Example:

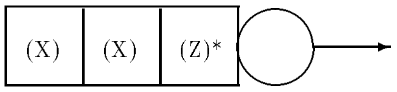
```
/DECLARE/ QUEUE A, B, C; CLASS X, Y, Z;

& ...

/STATION/ NAME = A; INIT = 1; SCHED = FIFO;
```



LAST FIRST

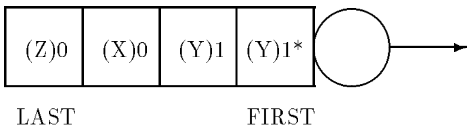
* = customer being served

```
/STATION/ NAME = B; INIT(X) = 2; INIT(Z) = 1;
          SCHED = LIFO,PREEMPT;
```



LAST FIRST

```
/STATION/ NAME = C; INIT(Y) = 2; INIT = 1;
          TYPE =MULTIPLE(3);
          PRIOR(Y) = 1; SCHED = PRIOR;
```



* = customer being served

4.3 Station

4.3.1 Single server station

A server may be regarded as a processor performing a service for customers. A service is comprised of work demands and/or object manipulation operations which are described in the SERVICE parameter of the station.

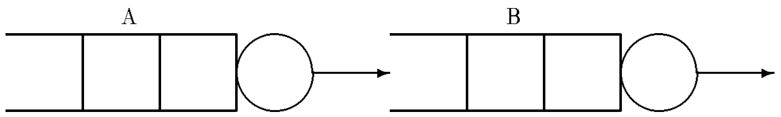
At service completion time the customer leaves the station and is forwarded to another station according to the probabilistic routing rules defined in the TRANSIT parameter. The server then takes the next customer waiting, if any. If no customer is waiting, the server becomes idle.

A customer leaves a station only at service completion time unless its transition is forced by a TRANSIT or MOVE procedure. A server can handle only one customer at a time.

The service of one customer may be interrupted and resumed later if the scheduling algorithm of the server has been chosen with preemption, with quantum or processor sharing.

Example:

```
/DECLARE  QUEUE A, B;

/STATION/ NAME    = A;
          TRANSIT = B;
          SERVICE = CST(10);
```



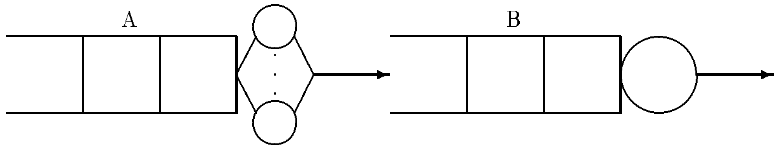
4.3.2 Multiple server station

The station includes several identical servers associated with the same queue. Several customers can therefore be served at the same time but a given customer is handled by only one server at a time.

The servers are equivalent: thus, when a customer requires a service, a selection of a server is made from the set of idle servers. The number of servers in a station may not be modified during solution time.

Example:

```
/DECLARE/ QUEUE A, B;

/STATION/ NAME    = A;
          TYPE    = MULTIPLE (2);
          SERVICE = CST(10);
          TRANSIT = B;
```

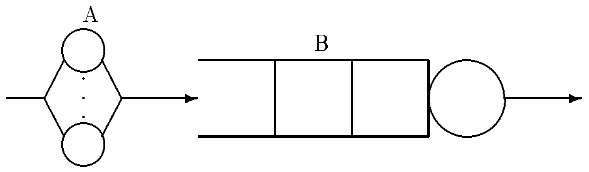


4.3.3 Infinite server station

The customers entering an infinite server station always find an idle server. Therefore, they never wait (the response time of such a station is equal to its service time). Such a station may be regarded as a pure delay.

Note that a multiple server station may be equivalent to an infinite one if the number of customers in the network can never exceed the number of servers of the station (for instance in the case of a closed network).

```
/DECLARE/ QUEUE A, B;

/STATION/ NAME    = A;
          TYPE    = INFINITE;
          SERVICE = CST (10.3);
          TRANSIT = B;
```



A customer entering into station A is sent to station B after a constant delay of 10.3 time units.

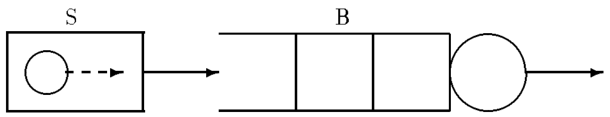
4.3.4 Source station

A source or generator station works as an infinite source of customers. The intervals between two successive customer generations are defined by the SERVICE parameter.

A source station may also be regarded as a simple server station which has been initialized with an infinite number of customers. When a customer has completed its service in this station it is sent in the network according to the routing rules defined by the TRANSIT parameter of the SOURCE station. The class of the generated customers must be explicitly specified at transition time, using the TRANSIT parameter. Forced transitions (using the TRANSIT procedure) of the current customer from a source station are not allowed.

The queue associated with a source station may not receive customers from other stations. In fact, it contains only the next customer to be emitted.

```
/DECLARE/ QUEUE S,B; CLASS X, Y;

&          ...

/STATION/ NAME    = S;
          TYPE    = SOURCE;
          TRANSIT = B,X;
          SERVICE = CST (20);
```



A customer is created every 20 time units and is sent to station B with class X.

4.4 Service

The service of a customer in a station starts as soon as a server is allocated to this customer. The service performed is defined by the SERVICE parameter of the station. It is made up of work demands and/or of object manipulation operations described by means of QNAP2 language procedures and QNAP2 algorithmic language.

4.4.1 Work demands

The work demand procedures specify the amount of work requested by a customer in the station. This amount of work may be a constant value (procedure CST) or a random value having a given distribution (procedures EXP, HEXP, ERLANG,...).

The work demand implies a service time which may depend on the server instantaneous service rate as specified by the RATE parameter of the station.

The work demands and the service rate must be expressed with coherent units so that a unique time unit may be determined for the network. For example:

- in a CPU station the work demands may be expressed in mega-instructions and the service rate in MIPS.
- in a CHANNEL station the work demands may be expressed in Kilo-bytes and the service rate in Kilobytes/s.

The default value of the service rate is 1 (i.e. 1 work unit per time unit). In this case work demands are expressed in time units.

All the random variables used in the work demands are assumed to be independent.

The completion of the service time of a customer currently in progress can be forced by executing a FREE procedure on this customer. The current service time is then interrupted (and will not be resumed), and the customer proceeds with the next operation specified in its service description.

4.4.2 Object manipulation operations

During a service, a customer may perform various operations on objects (customers, flags, resources or semaphores):

- force the transition of a customer into a station (TRANSIT, MOVE, BEFCUST and AFTCUST procedures),
- ask for a semaphore pass grant or give a semaphore pass grant (P and V procedures),
- request or release a resource (P and V procedures),
- test or set the state of flags (WAIT, WAITAND, WAITOR, SET, UNSET procedures),
- free a waiting customer (FREE procedure),
- create new customers (NEW function with parameter CUSTOMER),
- define a join operation between a customer and its siblings (JOIN and JOINC procedure),
- start and stop timers (SETTIMER:*keyword* procedures),
- enable and disable exceptions (SETEXCEPT:*keyword* procedures).

To be able to execute these operations a customer must hold a server, but these operations do not consume any processing time. Thus, for instance, resource allocation, semaphore testing or customer creation are instantaneous operations on the time scale of the model.

The P or WAIT operations may force a customer to wait for a resource, a semaphore or a flag. A waiting customer can no longer be processed. The server remains inactive and can not accept work demands nor execute service procedures (P, V,...). The customer is said to be blocked. The service continues (is resumed) as soon as the cause that produced the blocked state disappears (see “service suspension”, below).

A blocked customer keeps its server. It is considered by QNAP2 as still being served. The server remains inactive but occupied and can be assigned to a new customer only by means of the general allocation mechanisms (see “Server allocation”, below).

4.4.3 Service Completion

The service of a customer is completed:

- when all the work demands and the operations defined in the service have been completed. The customer leaves the station and is forwarded according to the TRANSIT parameter of the station (note that the expressions defining the queue and class destinations as well as the transition probabilities are evaluated only once at initiation time; dynamic transitions can be specified with the procedures MOVE, TRANSIT, BEFCUST and AFTCUST).
- if the customer explicitly asks to leave the station during its service or if another customer forces it to leave (refer to the procedures MOVE, TRANSIT, BEFCUST and AFTCUST).

In the latter case the service is interrupted and cannot be resumed, regardless of the remaining actions or requests.

Note:

At service completion (or at service suspension) time a customer does not release the resources that it may own. If a customer was blocked (waiting on a flag, a resource or a semaphore) a forced transition will not cancel the demands the customer posted.

4.4.4 Service Suspension

Execution of a service may be interrupted for various reasons:

- the server is preempted by the arrival of a customer with a higher priority level or by a quantum expiration,
- the customer is blocked:
  - the customer goes waiting (or is set to wait) for a resource, a semaphore or a flag (P,WAIT),
  - one of the resources of the customer is preempted.

As soon as the customer recovers the server or the resource it needs, its service restarts from the point of interruption.

Example:

Server preemption

```
/DECLARE/ QUEUE A;
          CLASS X, Y;

/STATION/ NAME      = A;
          SERVICE   = CST (10);
          SCHED     = PRIOR, PREEMPT;
          PRIOR (X) = 1;
          PRIOR (Y) = 2;
```

Consider the following sequence: at time 0 a class X customer starts being served by station A. At time 3 a class Y customer arrives. This customer preempts the server. X service is suspended and resumes at Y service completion time (= 13).

```
(X) ******|_ suspension______ **************
          |                   |
(Y) _____ ********************>_____________
          |                   |
      preemption             resume
    .     .   .         .     .   .         .
    0     3   5        10    13 v 15        20
```

4.5 Server Allocation

The presence of a customer in a queue corresponds to the allocation of a server (whatever its state may be).

When a server is allocated to a customer this customer is said to be served (even if its service is blocked for some reason). If no server is allocated to a customer (or as soon as the server has been preempted) the customer is said to be waiting. The server allocation mechanism is defined by the SCHED parameter of the station.

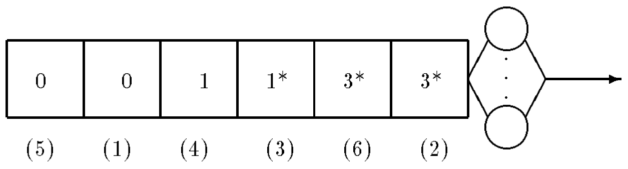
4.5.1 Station without quantum or preemption

A server is allocated only after the departure of the current customer. When a customer enters a station, and if an idle server exists, the server is immediately allocated to this customer (all the servers are equivalent in a station). If all the servers are busy at customer arrival time, then the customer is set to wait.

When a customer leaves the station, at service completion time or when forced by another customer, the freed server is immediatly allocated to the first waiting customer (scanning from the head of the queue). The servers are allocated independently of the other requests posted by the customer (thus a customer waiting for a resource or a semaphore remains candidate for a server).

Example:

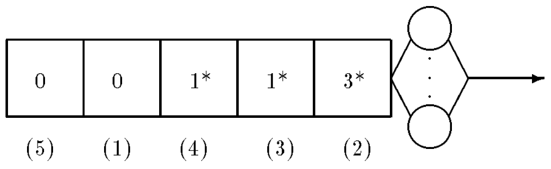
```
/DECLARE/ QUEUE A;

/STATION/ NAME  = A;
          TYPE  = MULTIPLE (3);
          SCHED = FIFO, PRIOR;
```



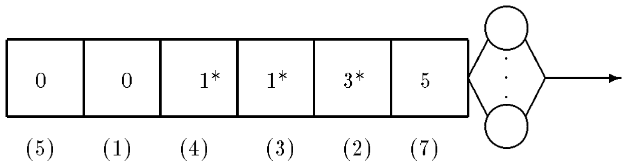
(arrival order)

* = customer holds a server

If customer (6) completes its service first, the queue becomes:



A new customer (7) enters the station:



Before being served, the customer (7) must wait for a service completion (since SCHED has not been specified with PREEMPT).

4.5.2 Station with preemption

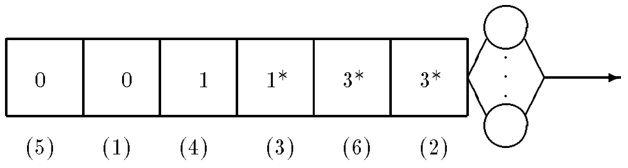
In a single server station with preemption (SCHED = PRIOR, PREEMPT or SCHED = LIFO, PREEMPT) the server is always detained by the first customer in the queue. As soon as a waiting customer reaches the head of the queue, the server is preempted from the customer being served and immediately allocated to the new customer. This preemption may occur under the following circumstances:

- SCHED = LIFO, PREEMPT : a new customer arrives (any customer),
- SCHED = PRIOR, PREEMPT : a new customer arrives having a higher priority level than the customer being served or in the case of a modification of the priority of the waiting customers (PRIOR procedure).

In the case of a multiple server station, the n servers are always allocated to the first n customers in the queue. If a customer arriving at the station or moved after a priority modification enters the set of the first n customers it then preempts the server previously allocated to the customer n (i.e. the customer which has the lower priority).

Example:

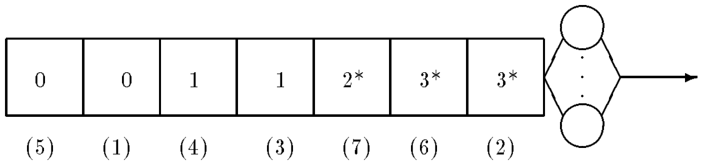
```
/DECLARE/ QUEUE A;

/STATION/ NAME  = A;
          TYPE  = MULTIPLE (3);
          SCHED = PRIOR, PREEMPT;
```



(arrival order)

* = customer holds a server

If a customer (7) arrives having a priority level of 2 the queue becomes:



Customer (7) preempts the server from customer (3).

4.5.3 Station with quantum allocation

In a single server station using a quantum allocation scheduling algorithm (SCHED = QUANTUM), the server is always allocated to the first customer in the queue. When this customer has worked for a period equal to the quantum it is forced to the end of the queue and the server is allocated to the next customer in line.

In the case of a multiple server station this mechanism applies for the first n customers.

Note:

- the quantum includes only real work: i.e. a customer that is blocked by other requests, and therefore is receiving no service from the server, stops the quantum decounting mechanism,
- the quantum value may be different for each customer class (refer to the QUANTUM parameter),
- a same station may not use the QUANTUM scheduling together with the PREEMPT or PRIORITY scheduling.

Example:

```
/DECLARE/ QUEUE A, B;
          CLASS X, Y;

/STATION/ NAME       = A;
          INIT(X)    = 2;
          INIT (Y)   = 1;
          SCHED      = QUANTUM (2);
          SERVICE(X) = CST (5.);
          SERVICE(Y) = CST(4.);
```

Three customers are in station A (2 class X customers and 1 class Y customer). The first services of station A are:

```
       service
  (X) ********|______ |______ ********|______ |______ ****>______
              |       |       |       |       |       |   |transition
  (X) _______ ********|______ |______ ********|______ |__ ****>___
              |       |       |       |       |       |       |transition
  (Y) _______ |______ ********|______ |______ ********>__ |_______
                                                      |transition
 TIME .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
```

Example:

```
/DECLARE/ QUEUE B;
          CLASS X,Y;

/STATION/ NAME       = B;
          INIT(X)    = 2;
          INIT(Y)    = 1;
          SCHED      = QUANTUM;
          QUANTUM(X) = 2;
          QUANTUM(Y) = 4;
          SERVICE(X) = CST(5.);
          SERVICE(Y) = CST(4.0);
```

The service quanta are different for class X and class Y customers.

```
   (X) ********|______ |______ |______ ********|______ ****>____
               |       |       |       |       |       |   |transition
   (X) _______ ********|______ |______ |______ ********|__ ****>__
               |       |       |       |       |       |   |transition
   (Y) _______ |______ ****************>______ |______ |__ |_____
                                       |transition
TIME .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
     0   1   2   3   4   5   6   7   8   9   10  11  12  13  14
```

4.5.4 Multiple server station with concurrency sets

Concurrency sets are defined with the EXCLUDE option of the SCHED parameter.

A customer entering a station is assigned a concurrency set, according to its class and/or a probability. A customer belongs to one concurrency set at most. It can belong to no concurrency set at all.

Two customers belonging to the same concurrency set may not be served simultaneously, even if there is an idle server. Note that this is meaningful only for multiple server stations (and infinite stations).

A customer belonging to no concurrency set is not concerned: it can seize an idle server whatever customer the other servers are busy with.

When a customer cannot seize a server due to concurrency, a subsequent customer may seize it: the server is allocated to the next eligible customer.

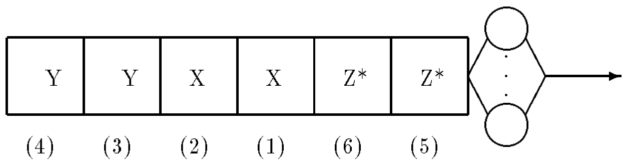
The concurrency sets can be used together with the other ordering mechanisms: FIFO, LIFO, PRIOR, PREEMPT, PS.

The concurrency sets are numbered in the order they are declared with the EXCLUDE option. The first set is numbered 1. The customer attribute CONCSETN yields the number of the concurrency set to which the customer belongs. It yields zero if the customer belongs to no concurrency set at all, or if no concurrency sets are defined for the station. This attribute can be used only in simulation. Note that it is assigned as soon as the customer enters the station.

Example:

```
      /DECLARE/ QUEUE A;
                CLASS X, Y, Z;

      /STATION/ NAME = A;
                TYPE = MULTIPLE (2);
                SCHED = PRIOR, PREEMPT, FEFS, EXCLUDE (X), (Y);
                PRIOR (Z) = 2;

                ...

                TRANSIT (NEW (CUSTOMER), A, X);
                TRANSIT (NEW (CUSTOMER), A, X);
                TRANSIT (NEW (CUSTOMER), A, Y);
                TRANSIT (NEW (CUSTOMER), A, Y);
                TRANSIT (NEW (CUSTOMER), A, Z);
                TRANSIT (NEW (CUSTOMER), A, Z);
```

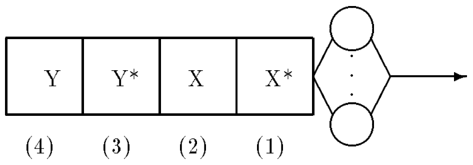
The preceeding statements yield the following ordering (assuming the queue was empty):



(arrival order)

* = customer holds a server

When the two class Z customers have completed their service, only one X and one Y customer can seize a server:



4.5.5 Resequencing station

A resequencing station is defined with the RESEQUENCE option of the SCHED parameter.

The customers are served in the resequencing station in the exact order that they departed from the reference station. The order in which they arrive at the resequencing queue has no importance.

A customer arriving out of sequence cannot be served until all its preceeding customers have arrived.

A customer arriving before its preceeding customers simply waits in the queue until all its preceeding customers arrived, even if there is an idle server. A customer arriving after its succeeding customers seizes the server.

Note:

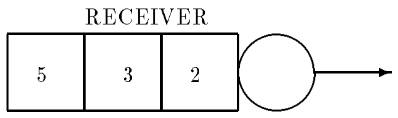
If one of the customers never arrives (because it was blocked somewhere, or destroyed), the other customers will not be served, and the resequencing station will hang forever.

Example:

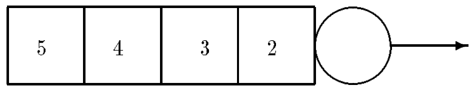
```
/STATION/ NAME = SENDER;
          TYPE = SOURCE;
          SERVICE = EXP (2);
          TRANSIT = NETW;

/STATION/ NAME = NETW;
          TYPE = INFINITE;
          SERVICE = EXP (1);
          TRANSIT = RECEIVER;

/STATION/ NAME = RECEIVER;
          SCHED = RESEQUENCE (SENDER);
          SERVICE = EXP (1.8);
          TRANSIT = OUT;
```

The NETW station mixes the order of the customers. The customers arriving at the RECEIVER station are ordered and served according to their sending order at the SENDER station.

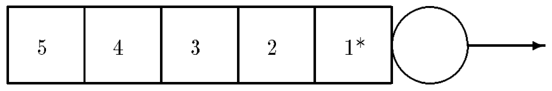
Assume that the SENDER station sent 6 customers, numbered from 1 to 6. Assume that customers 1, 4, 6 are still in the NETW station, whereas customers 2, 3, 5 arrived at the RECEIVER station. The queue contains:



Note that the server is idle. When customer 4 arrives, the queue becomes:



Note that 4 has been placed before 5. Now, 1 arrives:



Customer 1 seizes the server. When 1 has completed its service, the succeeding customers can proceed.

4.6 Customer Priority Level

Each customer has a priority level which may vary during its life in the model. This priority level is the CPRIOR attribute of the customer object.
The customer priority level is used to:

- order the customers in queues if these queues have a PRIOR scheduling (SCHED = PRIOR),
- determine the execution order of actions during a simulation run.

The priority level may only have positive or null values. The following mechanisms modify the priority level of customers.

4.6.1 PRIOR parameter

The PRIOR parameter of a station defines the priority level of the customers entering this station according to their classes. The customer classes for which no priority level is defined in a station keep their previous level.

Example:

```
/DECLARE/ QUEUE A;
          CLASS X,Y,Z;

/STATION/ NAME     = A;
          PRIOR(X) = 1;
          PRIOR(Y) = 3;
```

In station A the class X customers have a priority level of 1; class Y customers have 3 and class Z customers keep the one they had before (default value: 0).

Note:
The effect of a PRIOR parameter can be cancelled by forcing the priority level of a customer during a transition (see procedure TRANSIT, P, BEFCUST and AFTCUST).

4.6.2 PRIOR procedure

The PRIOR procedure modifies the priority level of the customer designated by the first parameter of the procedure argument list: (`PRIOR` (*[customer,] integer*)). If customer is omitted the customer to which PRIOR applies is the current customer (referenced by the variable CUSTOMER).
If the station uses a priority mechanism (SCHED = PRIOR) the customer whose priority level is modified is placed in a new position in the queue according to its new priority level (if the priority level given as an argument is equal to the previous priority level of the customer, the procedure is ineffective).
This procedure may induce a server allocation if the preemption mechanism is used. The modification of the priority level may also modify the ordering of actions in a simulation run (see Chapter "Simulation").

Example:

```
/DECLARE/ QUEUE A, B;
          CLASS X, Y, Z;
```

```
/STATION/ NAME     = A;
          SCHED    = PRIOR, PREEMPT;
          PRIOR(X) = 1;
          PRIOR(Y) = 2;
          PRIOR(Z) = 3;
          SERVICE  = BEGIN
                       CST(10);
                       PRIOR(4);
                       CST(1);
                     END;
```

During the first 10 time units the customer may be interrupted by an incoming customer. During the last time unit of the service the customer may not be interrupted since its priority level is higher than any other customer priority level.

Note:
If the priority level of a customer is modified the priority level of the requests it may have posted on semaphores or resources is not modified; this also applies for the priority level of the descendant customers it may have created (function call NEW(CUSTOMER)).

4.6.3 TRANSIT procedure

The TRANSIT procedure forces the priority level of a customer entering a station. If this mechanism is used, the PRIOR parameter of the station is not considered for the customer.

Example:

```
/DECLARE/ QUEUE A,B;
          CLASS X, Y;
          INTEGER PR;

/STATION/ NAME       = A;
          TRANSIT    = B;
          SERVICE(X) = BEGIN
                         CST(1);
                         TRANSIT(B, 3);
                       END;
          SERVICE(Y) = CST(2);

/STATION/ NAME       = B;
          PRIOR (X)  = 1;
          PRIOR (Y)  = 2;
```

In station B class X customers coming from station A have a priority level of 3 and class Y customers have 2 (determined by the PRIOR parameter of the station). The class X customers not coming from A have a priority level equal to 1.

4.6.4 BEFCUST and AFTCUST procedures

The BEFCUST and AFTCUST procedures may force the priority level of a customer entering a station if the priority parameter is present in the parameter list of the procedure. If the destination queue has a priority scheduling discipline, an error occurs if the specified priority level conflicts with the specified placement.

4.7 Customer Creation

A customer may dynamically create other customers during a service by means of the NEW function with the CUSTOMER argument: NEW(CUSTOMER) creates an object of type CUSTOMER and returns a reference to this object.

A newly created customer is initialized with the class and the priority level of its father, but this customer is not sent to a queue (CQUEUE attribute = NIL): it must be sent explicitly using the TRANSIT procedure.

- The FATHER attribute of customer points to the customer that created it (the value of this attribute is NIL for a customer created by a source station or at initiation time by the INIT parameter of the /STATION/ command).
- The SON attribute of the current customer points to the last created descendant.
- The SONNB function returns the number of sons created by a customer.
- The REFSON function returns a reference to the n^{th} alive son.

Example:

```
/DECLARE/ QUEUE A, B;
          CLASS X, Y;
          REF CUSTOMER C;

/STATION/ NAME        = A;
          SERVICE (X) = BEGIN
                          CST (1);

                          & descendant creation
                          & with C.CLASS = X
                          & and C.PRIOR = 3

                          PRIOR(3);
                          C:= NEW(CUSTOMER);

                          & C is sent to B
                          & with class Y

                          TRANSIT (C, B, Y);
                        END;
```

4.8 JOIN and JOINC procedures

Syntax:

`JOIN` [(*n* | *customer_list*)]
`JOINC` (*customer* [, *n* | *customer_list*])

Semantics:

The JOIN procedure causes the current customer to wait until some of its sons have been destroyed (i.e., transited to OUT or destroyed by a SPLIT or FISSION mechanism).

- If there is no parameter, the join operation concerns all the sons created by the current customer.
- When used with an integer parameter *n*, the current customer waits until *n* sons have been destroyed.
- When used with a explicit *customer_list*, the current customer waits until all the specified sons have been destroyed.

The JOINC procedure is similar to JOIN, except that it applies to the specified *customer* rather than the current customer. JOINC is used to force another customer to wait until its sons have been destroyed.

4.9 Customer Forced Transition

4.9.1 TRANSIT procedure

Syntax:

`TRANSIT` ([*customer*,] *queue* [, *class*] [, *priority_level*])

The TRANSIT procedure forces the transition of the specified *customer* into the specified *queue* with the specified *class* and integer *priority_level*. If *customer* is omitted the TRANSIT procedure applies to the current customer (referenced by the predefined variable CUSTOMER).

If the customer is being served, it loses its server. Its service is terminated and cannot be resumed. If the destination queue is OUT, the *customer* is destroyed.

Example:

```
/DECLARE/ QUEUE A, B, C; CLASS X, Y;

/STATION/ NAME    = A;
          TRANSIT = B, 1, C, Y, 1;

& this description is also equivalent to:

/STATION/ NAME    = A;
          SERVICE = BEGIN
                      EXP (3);
                      IF DRAW (0.5)
                         THEN TRANSIT (B)
                      ELSE TRANSIT (C, Y);
                      & ** not reached **
                    END;
```

Note:

- A blocked customer or a customer holding resources may not be destroyed.
- Statements following a TRANSIT of the current customer will never be executed.

4.9.2 MOVE procedure

Syntax:

`MOVE` (*queue1*, *queue2* [, *class*] [, *priority_level*])

Semantics:

The MOVE procedure forces the transition of the first customer of station *queue1* into the station *queue2* under the similar conditions as the TRANSIT procedure. If *queue1* is empty the procedure call is ignored.

Example:

```
/DECLARE/ QUEUE A, B, C;
          CLASS X, Y;

/STATION/ NAME    = A;
          SERVICE = BEGIN
                      CST (1);
                      TRANSIT (NEW(CUSTOMER), B, X, 3);
                      MOVE (C, A);
                    END;
```

During its service in A a customer creates a son and sends it to queue B with class X and a priority level of 3. Then it forces the first customer in the C queue to transit to A.

4.9.3 BEFCUST and AFTCUST procedures

Syntax:

BEFCUST (*customer1*, *customer2* [, *class*] [, *priority_level*])
AFTCUST (*customer1*, *customer2* [, *class*] [, *priority_level*])

Semantics:

The BEFCUST procedure forces the transit of *customer1* to the queue containing *customer2*. *customer1* is placed before *customer2*, with the specified *class* and *priority_level*.

The AFTCUST procedure has the same properties, except that *customer1* is placed after *customer2*.

If *customer2* is not in a queue, the execution of the procedures BEFCUST or AFTCUST causes a simulation error.

If the queue of customer2 has a priority queueing discipline (SCHED = PRIOR), a consistency check is performed in order to verify that the forced placement does not conflict with the queueing discipline. A conflict causes a simulation error.

4.10 Customer split and match

Syntax:

```
SPLIT [(class_list)] =     { ( {queue, class, count} [, ...] ) [prob] } [, ...];
MATCH =                    { (queue [, class_list]) :
                           ( {class, count } [, ...] ) result_class
                           [ : PRIOR (integer) ] } [, ...];                     |
                           { (queue [, class_list]) :
                           ( {class, count } [, ...] ) result_class
                           [ : WEIGHT (real) ] } [, ...];
FISSION [(class_list)] =   { ( {queue, class, count} [, ...] ) [prob] } [, ...];
FUSION =                   ( {class, count } [, ...] ) result_class
                           [ : PRIOR (integer) ] } [, ...];                     |
                           ( {class, count } [, ...] ) result_class
                           [ : WEIGHT (real) ] } [, ...];
```

Semantics:

Customers can be split and joined up via the following mechanisms:

SPLIT: a single customer is broken up into any number of customers. The resulting "pieces" can belong to various classes. The origin of the "pieces" is kept for later join up with MATCH.

MATCH: several customers representing "pieces" resulting from a previous SPLIT are joined up to form a single customer.

FISSION: same as SPLIT, except that the origin of the "pieces" is not remembered.

FUSION: several customers are joined up to form a single customer. The origin of the joined customer is ignored.

Note:

1. MATCH must be used in conjonction with SPLIT. FISSION and FUSION may be used without restriction.
2. All these operations destroy the input customer(s) and create new customers. There is no father/son relationship between the input and output customers. The destroyed customers are considered as if they had performed a TRANSIT to the OUT queue. This may cause a parent customer waiting with JOIN or JOINC to be freed.
3. These operations work only with pure CUSTOMER objects. Sub-types of CUSTOMER are split into CUSTOMERs, not into sub-type objects.

Customers resulting from a SPLIT can be split again. The origin of all successive SPLITs is kept.

A MATCH operation needs not join up *all* pieces that resulted from a SPLIT. Partial join ups can be performed.

Several SPLIT and FISSION possibilities may be specified. The selection is performed with a probabilistic switch.

Several MATCH and FUSION possibilities may be specified. When several join ups are possible simultaneously, the selection is performed either via integer priority levels (PRIOR option), or via a probabilistic switch (WEIGHT option).

SPLIT and FISSION replace the TRANSIT parameter of the splitting station. The customers perform their service normally. They are split *after* completion of their service.

Join ups are performed at the *input* of a station. The resulting customer performs its service normally.

When only part of the customers required for a join up have reached the station, they wait *before* the station. They are considered as being in no queue at all. They are not taken into account in the queue statistics. When all required customers have arrived, the join up is performed immediately. The input customers are destroyed and the resulting customer enters the station.

Several MATCH operations can proceed simultaneously on the same station. The "pieces" corresponding to different origin customers wait separately that all the required customers have arrived.

Note:

A MATCH is performed only when all the required "pieces" have arrived. If a "piece" is lost (blocked somewhere or destroyed), the other "pieces" will wait forever before the matching station.

Example:

In a manufacturing plant, iron sheets must be cut into 12 pieces, 4 of size A and 8 of size B. A and B parts are tooled separately. At the end, one A part and two B parts must be welded together to form a C part.

```
/DECLARE/ QUEUE CUT, TOOLA, TOOLB, WELD;
          CLASS SHEET, A, B, C;

/STATION/ NAME = CUT;
          INIT (SHEET) = 100;
          SERVICE (SHEET) = CST (10);
          SPLIT (SHEET) = (TOOLA, A, 4, TOOLB, B, 8);

/STATION/ NAME = TOOLA, TOOLB;
          SERVICE = CST (1);
          TRANSIT = WELD;

/STATION/ NAME = WELD;
          MATCH = (CUT, SHEET) : (A, 1, B, 2) C;
          SERVICE = CST (2);
          TRANSIT = OUT;
```

Note that in this example, A and B parts welded together *must* come from the same sheet. If this is not a requirement, then the fission/fusion mechanism can be used instead. FUSION has a slightly different syntax as the origin information does not apply:

```
          FUSION = (A, 1, B, 2) C;
```

4.11 Semaphores

During its service, a customer may ask a pass grant to a semaphore. If the request is refused the customer is blocked until the grant is given. A customer may also force another customer (waiting or being served) to ask for a pass grant.

A semaphore station consists of a queue and a counter. The counter is the number of pass grants available if positive, and the number of customers waiting if negative.

4.11.1 Semaphore creation

A semaphore is statically created by means of the /STATION/ control statement with the following parameters:

```
TYPE = SEMAPHORE, [MULTIPLE ( integer ) ]
```

The integer value is the initial value of the counter (default value: 1). Only the PRIOR parameter and SCHED parameters may be specified in the /STATION/ command.

A semaphore may be dynamically created as an instance of an object of type QUEUE or one of its sub-types.

4.11.2 P procedure

Syntax:

P ([*customer,*] *queue* [, *class*] [, *priority_level*])

Semantics:

The P procedure forces the *customer* to ask for a pass grant to the semaphore *queue*. If *customer* is omitted the pass grant is asked for the current customer (referenced by the predefined variable CUSTOMER)).

If the counter of the semaphore is positive the P operation does not affect the customer; the semaphore counter is decremented by 1. If the semaphore counter is null or negative the grant is refused. A new customer representing the request is then created and sent to the semaphore queue. The request is posted with the specified *class* and *priority_level*. If these parameters are omitted, the request is posted with the class and priority level of the requesting customer.

A blocked customer requesting a pass grant is freed if a pass grant becomes available or if a FREE procedure is executed on this customer.

Note:

- A single customer may be forced to ask for several grants on different or on the same semaphore.
- Preemption is meaningless in the case of a semaphore: a semaphore gives only pass grants.

4.11.3 V procedure

Syntax:

V (*queue*)

Semantics:

The V procedure produces a pass grant for the semaphore *queue*. If the semaphore counter is null or positive the semaphore counter is simply incremented by 1 (the grant may be used later).

If the semaphore counter is negative (meaning that some customers are waiting) the request at the head of the queue is cancelled and the customer that posted this request receives the pass grant and thus may proceed with its service, if it is not waiting for other reasons.

Note:

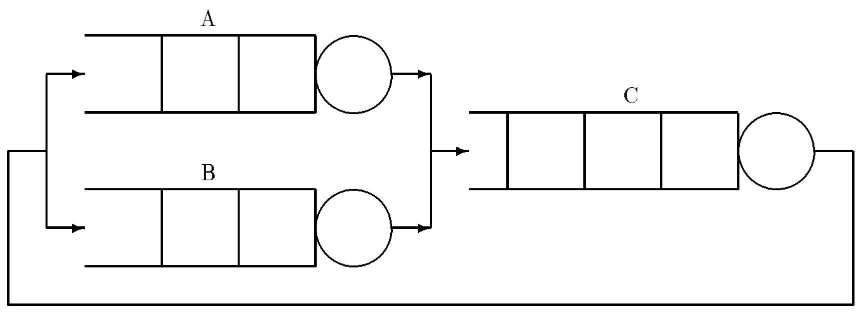
A customer may perform a V procedure on a semaphore even if it never performed a P procedure on this semaphore (this is not true in the case of P and V procedures on resources).

Example:

Finite capacity queue

```
/DECLARE/ QUEUE A, B, C, S;

/STATION/ NAME    = A;
          INIT    = 20;
          TRANSIT = C;
          SERVICE = BEGIN
                      EXP(5);
                      P(S);
                    END;

/STATION/ NAME    = B;
          TRANSIT = C;
          SERVICE = BEGIN
                      EXP(8);
                      P(S);
                    END;

/STATION/ NAME    = C;
          TRANSIT = A,1,B,1;
          SERVICE = BEGIN
                      CST(10);
                      V(S);
                    END;
/STATION/ NAME = S;
          TYPE = SEMAPHORE,MULTIPLE(3);
```

The station C has a queue capacity limited to three customers (including the one being served). When the queue is full the customers may not enter the station and have to stay in their previous station A or B.



Note:

This example is shown as an illustration only. Finite capacity queues are directly implemented in QNAP2 with the CAPACITY parameter of the /STATION/ command.

4.11.4 PMULT and VMULT procedures

Syntax:

PMULT ([*customer*,][*list_of_priority_levels*,]*list_of_queues*[,*list_of_classes*][,*list_of_integers*])
VMULT ([*customer*,]*list_of_queues*[,*list_of_integers*])

Semantics:

The PMULT and VMULT procedures apply to queues whose type is semaphore and/or resource.

The PMULT procedure does simultaneously and in an indivisible way a whole set of P requests to the semaphores and/or resources in the *list_of_queues* argument. The execution of this procedure is strictly equivalent to an indivisible succession of P operations that will never be interrupted and that will be totally fulfilled, even if some of these operations are blocking.

The VMULT procedure does simultaneously and in an indivisible way a whole set of V operations to the semaphores and/or resources in the *list_of_queues* argument.

The *list_of_integers* argument is the list of the numbers of P or V operations performed to the corresponding queue in the *list_of_queues* argument. The *list_of_priority_levels* and *list_of_classes* arguments of PMULT procedure are the lists of the classes and priority levels with which the corresponding requests are performed.

Note:

PMULT and VMULT are not implemented in the QNAP2 releases anterior to V9.1 .

Example:

```
SIMULOG   ***  QNAP2  ***  ( 01-04-95  ) V 9.2
(C)  COPYRIGHT BY CII HONEYWELL BULL AND INRIA, 1986


    1 & This model is to show in a simple way the difference between
    2 & PMULT and a succession of P
    3 & In the first part of the model, the current customer in the
```

```
& "double" station performs a request of pass grant to "sem2"
& only if its first request (to "sem1") has succeeded.
& Whereas both requests are performed simultaneously in the second
& part of the model (with PMULT)

/CONTROL/ OPTION=SOURCE,RESULT,TRACE;

/DECLARE/ QUEUE double,simple1,simple2;
          QUEUE sem1,sem2;

/STATION/ NAME=double;
          INIT=10;
          TYPE=SINGLE;
          TRANSIT=OUT;
          SERVICE=BEGIN
            P(sem1); P(sem2);        & if the customer is blocked by
                                     & its first request, he does not
                                     & perform the second
            EXP(2);
            V(sem1); V(sem2);
          END;

/STATION/ NAME=simple1;
          INIT=10;
          TYPE=SINGLE;
          TRANSIT=OUT;
          SERVICE=BEGIN
            P(sem1);
            EXP(1);
            V(sem1);
          END;

/STATION/ NAME=simple2;
          INIT=10;
          TYPE=SINGLE;
          TRANSIT=OUT;
          SERVICE=BEGIN
            P(sem2);
            EXP(1);
            V(sem2);
          END;

/STATION/ NAME=sem1;
          TYPE=SEMAPHORE,SINGLE;

/STATION/ NAME=sem2;
          TYPE=SEMAPHORE,SINGLE;

/CONTROL/ TMAX=100;

```

```
    54 /EXEC/ SIMUL;

 *** SIMULATION ***


...


 ------------------------------------------------------------------------------
  CUSTOMER   QUEUE   CLASS   PRIOR    NB           OPERATION
 ------------------------------------------------------------------------------

 TIME: 0.000
 >      1  double                0    10   P sem1     VALUE:      0
 >      1  double                0    10   P sem2     VALUE:      0
 >      1  double                0    10   DELAY       1.801 UNTIL        1.801
 >     11  simple1               0    10   P sem1     VALUE:      0
 >     21  simple2               0    10   P sem2     VALUE:      0


...


 TIME: 26.618
 >     20  simple1               0     1   V sem1     VALUE:      1
 >     20  simple1               0     0   ==> OUT

 TIME: 100.000
 > TIMER TSYSTMAX                          IS JUST ACTIVATED
 ... TIME =          100.00
 *******************************************************************
 *   NAME   *  SERVICE * BUSY PCT *  CUST NB * RESPONSE *  SERV NB *
 *******************************************************************
 *          *          *          *          *          *          *
 * double   * 2.644    *0.2644    * 1.480    * 14.80    *        10*
 *          *          *          *          *          *          *
 * simple1  * 2.662    *0.2662    * 1.583    * 15.83    *        10*
 *          *          *          *          *          *          *
 * simple2  * 1.233    *0.1233    *0.8410    * 8.410    *        10*
 *          *          *          *          *          *          *
 * sem1     *    0.    *    0.    *0.2644    * 1.392    *        19*
 *          *          *          *          *          *          *
 * sem2     *    0.    *    0.    *0.8408E-01* 1.051    *         8*
 *          *          *          *          *          *          *
 *******************************************************************
 ... END OF SIMULATION ...


              MEMORY USED:       6405 WORDS OF 4 BYTES
               (  0.64  % OF TOTAL MEMORY)
    55
    56 /RESTART/     & start another model
1

  SIMULOG   ***  QNAP2  ***  ( 01-04-95  ) V 9.2
  (C)  COPYRIGHT BY CII HONEYWELL BULL AND INRIA, 1986


     1
     2 /CONTROL/ OPTION=SOURCE,RESULT,TRACE;
     3
     4 /DECLARE/ QUEUE double,simple1,simple2;
     5           QUEUE sem1,sem2;
     6
     7 /STATION/ NAME=double;
     8           INIT=10;
     9           TYPE=SINGLE;
    10           TRANSIT=OUT;
    11           SERVICE=BEGIN
    12             PMULT((sem1,sem2),(1,1));       & both requests are performe
    ==>  d
    13             EXP(2);
    14             VMULT((sem1,sem2),(1,1));
    15           END;
    16
    17 /STATION/ NAME=simple1;


...


    45 /EXEC/ SIMUL;

 *** SIMULATION ***


...


 -------------------------------------------------------------------------------
  CUSTOMER   QUEUE   CLASS   PRIOR    NB           OPERATION
 -------------------------------------------------------------------------------

 TIME: 0.000
 >      1  double                0    10   PMULT ON QUEUES:
          SEMAPHORE sem1         1 REQUESTS
          SEMAPHORE sem2         1 REQUESTS
 ==> CUSTOMER UNBLOCKED
 >      1  double                0    10   DELAY       1.801 UNTIL        1.801
 >     11  simple1               0    10   P sem1     VALUE:      0
>     21  simple2               0    10   P sem2     VALUE:      0


...


TIME: 30.486
>     20  simple1               0     1   V sem1     VALUE:      1
>     20  simple1               0     0   ==> OUT

TIME: 100.000
> TIMER TSYSTMAX                          IS JUST ACTIVATED
... TIME =          100.00
********************************************************************
*   NAME   *  SERVICE * BUSY PCT *  CUST NB * RESPONSE *  SERV NB *
********************************************************************
*          *          *          *          *          *          *
* double   * 2.962    *0.2962    * 1.516    * 15.16    *        10*
*          *          *          *          *          *          *
* simple1  * 3.049    *0.3049    * 1.588    * 15.88    *        10*
*          *          *          *          *          *          *
* simple2  * 2.979    *0.2979    * 1.580    * 15.80    *        10*
*          *          *          *          *          *          *
* sem1     *    0.    *    0.    *0.2962    * 1.559    *        19*
*          *          *          *          *          *          *
* sem2     *    0.    *    0.    *0.2962    * 1.559    *        19*
*          *          *          *          *          *          *
********************************************************************
... END OF SIMULATION ...


             MEMORY USED:       6797 WORDS OF 4 BYTES
              (  0.68  % OF TOTAL MEMORY)
    46
    47 /END/
```

4.12 Resources

A resource is a station consisting of a queue and one or more resource units (or passive servers). During a service, a customer may require a resource unit. If the request is not satisfied, the customer is blocked until a resource unit is available.

4.12.1 Resource creation

A resource is statically created by means of the /STATION/ control statement with the following parameter:

```
TYPE = RESOURCE [, MULTIPLE (integer)];
```

The integer value specifies the number of customers that may share the resource simultaneously. Only the PRIOR parameter and SCHED parameters may be used for a resource.

4.12.2 P procedure

Syntax:
`P` ([*customer*,] *queue* [, *class*] [, *priority_level*])

Semantics:
The P procedure forces *customer* to request a resource unit from the resource *queue*. If customer is omitted the resource is requested by the customer being served (referenced by the variable CUSTOMER).

A new customer representing the request is created and sent to the queue associated with the resource station, with the specified *class* and *priority_level*. If these parameters are omitted, the request is posted with the class and priority level of the requesting customer.

If a resource unit is available it is allocated to the customer which can then proceed with its service. The statisfied request remains in the queue of the resource as long as this resource is kept by the customer. If no resource unit is available the requesting customer is blocked. The service of this customer is interrupted but the server on which it was working is not freed.

A blocked customer waiting for a resource is freed if a unit of resource becomes available.

4.12.3 V procedure

Syntax:
`V` ([*customer*,] *queue*)

Semantics:
The V procedure releases the resource unit allocated by *queue* to *customer*. If *customer* is omitted, the default is the curent customer (referenced by the variable CUSTOMER). If the queue associated with the resource contains waiting requests, the resource is allocated to another customer depending on the ressource discipline: FIFO, PRIOR...

Note:
A customer may not perform a V operation on a resource on which it did not perform a P operation (as opposed to semaphores).

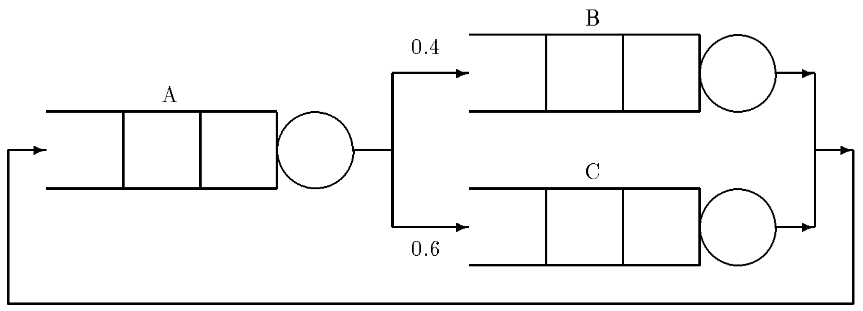
4.12.4 Server-resource equivalence

Servers and resources correspond to symmetrical views of synchronization. A model consisting of a set of servers processing customers may be represented by a symmetrical model made up of customers requesting resources.

The following example shows two versions of the same model to illustrate this duality.

Example:
Version 1

```
        /DECLARE/ QUEUE A, B, C;

        /STATION/ NAME    = A;
                  INIT    = 20;
                  TRANSIT = B,0.4,C;
                  SERVICE = EXP(5);

        /STATION/ NAME    = B;
                  TRANSIT = A;
                  SERVICE = EXP(3);

        /STATION/ NAME    = C;
                  TRANSIT = A;
                  SERVICE = EXP(2);
```



Example:
Version 2

```
/DECLARE/ QUEUE A,B,C, CUSTOM;

/STATION/ NAME = CUSTOM;
          TYPE = INFINITE;
          INIT = 20;
          SERVICE = WHILE TRUE DO BEGIN
                        P(A);
                        EXP(5);
                        V(A);
                        IF DRAW(0.4)
                        THEN BEGIN
                        P(B);
                        EXP(3);
                        V(B);
                     END
                     ELSE BEGIN
                        P(C);
                        EXP(2);
                        V(C);
                     END;
                  END;

/STATION/ NAME = A;
          TYPE = RESOURCE;

/STATION/ NAME = B;
          TYPE = RESOURCE;

/STATION/ NAME = C;
          TYPE = RESOURCE;
```

In the first version, there are three servers and the customers are processed according to the mechanisms of queueing network models.

In the second version, the CUSTOM station allocates a server to each customer. The processing of one customer is explicitly described: posession of resources A,B and C synchronizes these activities. The customer transitions correspond to P and V operations.

4.12.5 PMULT and VMULT procedures

These procedures are described in the preceding chapter, about the semaphores.

4.13 Flags

The flags may be used to synchronize customers with simpler tools than semaphores or resources. This mechanism is especially useful to describe parallel activities.

A flag has two possible states: "set" and "unset". The STATE attribute of a flag represents its current state (the TRUE value being the "set" state and the FALSE value being the "unset" state).

Customers manipulate flags by means of procedures and functions.

4.13.1 Flag creation

Syntax:

`NEW` (*flag*)

Semantics:

The NEW function dynamically creates a flag; the initial state of this flag is "unset". This NEW function returns a reference to the created flag. Flags may also be statically created within the /DECLARE/ command by declaring variables of type FLAG.

4.13.2 SET, RESET and UNSET procedures

Syntax:

`SET` (*flag*)
`UNSET` (*flag*)
`RESET` (*flag*)

Semantics:

The SET procedure puts the specified *flag* in the "set" state.

The UNSET and RESET procedures put the specified *flag* in the "unset" state.

4.13.3 WAIT, WAITAND and WAITOR procedures

Syntax:

`WAIT` ([*customer*,] *flag*)
`WAITAND` ([*customer*,] *flag_list*)
`WAITOR` ([*customer*,] *flag_list*)

Semantics:

The WAIT procedure performs a simple waiting condition. It forces *customer* to test the state of *flag*. If *customer* is omitted the default is the current customer (referenced by the variable CUSTOMER).

If the flag state is "unset" then the customer is forced to wait on the flag. While waiting it remains in its queue and if it was being served it keeps its server. The waiting customer is freed when the flag passes into the "set" state, or if a call to the FREE procedure occurs. If the flag is in the "set" state the WAIT procedure is ignored.

Note:

Customers waiting on a single flag are chained together according to a LIFO discipline: it is possible to scan the customers waiting on a flag using the following items:

- the LIST attribute of the flag references the last customer arrived,
- the LINK attribute of a customer references the next customer waiting on the same flag.

The WAITAND procedure performs a conjunct waiting condition. It causes *customer* (defaulting to the current customer) to wait until all the flags in *flag_list* are in the "set" state. The waiting customer is freed when all the flags are in the state "set", or when a call to the FREE procedure occurs.

The WAITOR procedure defines a disjunct waiting condition. It causes the *customer* (defaulting to the current customer) to wait until one of the flags in *flag_list* is in the "set" state. The waiting customer is freed when one of the flags is in the "set" state, or if a call to the FREE procedure occurs.

4.14 Exceptions

Exceptions are QNAP2 objects used to catch special events: such as external signals sent by the operating system (e.g., user interrupt, error condition). QNAP2 uses predeclared exception objects to handle internal events, such as start/end of the simulation

Each exception object is assigned a procedure, called the *handler*. The handler is called automatically by QNAP2 when the exception is raised. The handler can perform any algorithmic language operation, except work demands or blocking synchronizations: a handler is not allowed to spend simulation time or become suspended, but it can operate on customers, queues, flags, exceptions and timers.

4.14.1 Exception creation

An exception object may be created statically with the /DECLARE/ command or dynamically with the NEW function.

Exception objects are parameterized. The optional parameter is the name of the handler procedure. This allows to assign the handler procedure upon creation of the exception object.

Example:

```
    /DECLARE/

    PROCEDURE MESSAGE;
    BEGIN
      PRINT ("Exception raised");
    END;

   EXCEPTION (MESSAGE) INTR;
```

When used with external events, an exception object must be *connected* to the external event. Once connected, the exception is automatically raised when the external event occurs. The applicable external events are operating-system dependent. Refer to the QNAP2 installation guide for the list of external events available on your system.
Exception objects have the following attribute:

DEFHANDLER: reference to the handler procedure.

The predeclared exceptions are the following:

SIMSTART (resp. SIMSTOP) is raised automatically at the beginning (resp. end) of the simulation. The default handler is empty.

Note:

As opposed to the ENTRY and EXIT parameters of the /CONTROL/ command, SIMSTART and SIMSTOP are considered part of the simulation. When used with the replication method, the ENTRY and EXIT sequences are executed only once, before and after the whole set of simulations, whereas SIMSTART and SIMSTOP are raised once for each simulation. Furthermore, synchronization operations are not allowed in the ENTRY and EXIT sequences.

SIMACCUR is raised automatically when all the statistical variables for which a required accuracy has been specified (see section 5.4.4.2 "Accuracy control" on page 205) are satisfactory. The default handler is the STOP procedure. The user can replace the default handler to perform additional controls and decide whether the simulation be actually stopped.

Note:

If the user handler does not explicitely stop it, the simulation goes on. In this case, the SIMACCUR exception can be raised again later.

4.14.2 Exception manipulation

Exception objects are handled by the SETEXCEPT:*keyword* procedures.

Syntax:

```
SETEXCEPT:CONNECT (exception, signal_name [, procedure])
SETEXCEPT:DISCONNECT (exception)
SETEXCEPT:HANDLER (exception, procedure)
SETEXCEPT:MASK (exception)
SETEXCEPT:UNMASK (exception)
SETEXCEPT:LAUNCHTIMER (exception, delay)
SETEXCEPT:CANCELTIMER (exception)
```

where `exception` is an exception or a reference to an exception.

CONNECT is used to connect an exception to an external signal. `signal_name` is the symbolic name of the external signal, as a character string enclosed in double quotes. Leading and trailing blanks are ignored. The available signals depend on the operating system (refer to the installation guide). The optional `procedure` specifies the procedure handler to call when the exception is raised.

DISCONNECT is used to cancel a previous SETEXCEPT:CONNECT call. Handling of the signal is left to the operating system.

HANDLER is used to specify the procedure to call when the exception is raised. `procedure` must be the identifier of (or a reference to) a procedure with no argument.

Note:

CONNECT and DISCONNECT may not be used with the predeclared exceptions. HANDLER must be used to assign a procedure handler to these exceptions.

MASK and UNMASK: MASK is used to temporarily mask occurrences of an exception. The exception stays hidden until it is unmasked with UNMASK. MASK increments a counter, whereas UNMASK decrements it. The exception is considered as unmasked when the counter is null (initial value).

LAUNCHTIMER: when the exception is connected to an external timer signal, LAUNCHTIMER is used to schedule this timer for `delay` seconds. If the timer was already running, the previous setting is cancelled.

Note:

This feature should not be confused with timer objects described in the next section. An exception can be connected to a *real time* clock or to a *cpu time* clock. This time has nothing to do with the internal simulation time.

CANCELTIMER is used to cancel any previous LAUNCHTIMER setting.

Example:

```
/DECLARE/ EXCEPTION CPULIMIT;

PROCEDURE CLEANUP;
BEGIN
  PRINT ("CPU time limit expired");
  PRINT ("Cleaning up at ", TIME);
  SAVERUN ("suspend");
  STOP;
END;

/EXEC/
BEGIN
  SETEXCEPT:CONNECT (CPULIMIT, "CPU TIME", CLEANUP);
  SETEXCEPT:LAUNCHTIMER (CPULIMIT, 3600);
END;
```

The exception object CPULIMIT is connected to the external signal "CPU TIME", controlling the CPU time limit. Then, the CPU timer is launched for 3600 seconds. When the CPU time limit is reached, the operating system sends the "CPU TIME" signal to QNAP2. The exception is raised and the CLEANUP procedure is executed, in order to save the model and stop the simulation in a clean way.

4.15 Timers

Timers are QNAP2 objects used to model processes independently of the queueing network. QNAP2 uses predeclared timers to perform some simulation functions (e.g., PERIOD, TSTART, TMAX). The user can define his own timers. Timers can be used only in simulation.

Each timer is assigned a procedure, called the *handler*. The handler is called automatically by QNAP2 when the timer expires. The handler can perform any algorithmic language operation, except work demands or blocking synchronizations: a handler is not allowed to spend time or become suspended, but it can operate on customers, queues, flags, exceptions and timers.

Note:

Timers should not be confused with the operating-system timers that can be connected toi a QNAP2 exception (see previous section). QNAP2 timers operate on simulation time, not on real time.

4.15.1 Timer creation

A timer can be created statically in a /DECLARE/ command or dynamically with the NEW function.

Timer objects have the following attributes:

HANDLER is a reference to the handler procedure.

TIMPRIOR is the integer priority level.

STATE takes the following values:

0 when the timer is idle,
1 when it was activated by SETTIMER:ABSOLUTE,
2 when it was activated by SETTIMER:RELATIVE,
3 when it was activated by SETTIMER:CYCLIC, and
4 when it was activated by TRACKTIME.

ACTIARG is the value of the argument passed to the last activation procedure used.

Note:

1. ACTIARG should not be used to guess the next wake-up time of the timer: use the GETSIMUL:WAKETIME function instead.
2. All the timer attributes are read-only. They can be changed by SETTIMER:*keyword* procedures.

The predeclared timers are the following:

TSYSTMAX stops the simulation when TMAX is reached.

TSYSTSTR manages the starting period (TSTART parameter).

TSYSPERI is activated periodically (PERIOD parameter) to manage the TEST sequence.

TSYSTRACE manages the starting and ending times of the trace.

TSYSSPCT is used internally by the spectral method.

TSYSWDOG is used internally for QNAP2 debugging.

The user can read the attributes of the predefined timers and change some of them, although this is not recommended as it interferes with their normal operation.

4.15.2 Timer manipulation

Timer objects are handled by the SETTIMER:*keyword* procedures. These procedures can be used only during the simulation.

Syntax:

```
SETTIMER:SETPROC (timer, procedure);
SETTIMER:ABSOLUTE (timer, date);
SETTIMER:RELATIVE (timer, delay);
SETTIMER:CYCLIC (timer, delay);
SETTIMER:TRACKTIME (timer);
SETTIMER:CANCEL (timer);
```

where `timer` is a reference to a timer object.

SETPROC is used to assign a handler procedure to a timer. `procedure` must be the name of (or a reference to) a procedure with no argument.

ABSOLUTE schedules the timer for the specified simulation `date`, which must be greater than TIME.

RELATIVE schedules the timer for `TIME + delay`.

CYCLIC schedules the timer for `TIME + delay`. After activation, the timer is automatically rescheduled for the same duration.

TRACKTIME schedules the timer to track time changes. The timer is activated once just before time changes, i.e., after all simultaneous events. The timer is rescheduled immediately.

CANCEL cancels any previous activation operation and makes the timer idle.

Note:

1. ABSOLUTE, RELATIVE, CYCLIC and TRACKTIME override any previous setting. If the timer was already scheduled, the previous setting is cancelled.
2. Most of these operations are forbidden on predeclared timers. Refer to the Reference Manual.

4.15.3 Timer priority

Syntax:

```
PRIOR (timer, integer);
```

Semantics:
When several timers expire at the same simulation time, their handlers are activated one after the other. The timer priorities can be used to control the activation orders: the timer with the highest priority level is activated first.

The PRIOR procedure is used to set the integer priority level of a timer to a specific level.

4.15.4 Example

```
SIMULOG   ***  QNAP2  ***  ( 01-04-95  ) V 9.2
(C)  COPYRIGHT BY CII HONEYWELL BULL AND INRIA, 1986


/DECLARE/ QUEUE PRODUCE, PACKAGE;
          TIMER FAILURE;

/STATION/ NAME = PRODUCE;                         & Produce items
          TYPE = SOURCE;
          SERVICE = EXP (10);
          TRANSIT = PACKAGE;

/STATION/ NAME = PACKAGE;                         & Package items
          SERVICE = EXP (5);
          TRANSIT = OUT;

/DECLARE/ PROCEDURE RESTART;
BEGIN
  UNBLOCK (PACKAGE);
END;

/DECLARE/ PROCEDURE FAIL;
BEGIN
  BLOCK (PACKAGE);
  SETTIMER:SETPROC (FAILURE, RESTART);
  SETTIMER:RELATIVE (FAILURE, 100);
END;

/DECLARE/ PROCEDURE INITIMER;                     & Initialize timer

BEGIN
  SETTIMER:SETPROC (FAILURE, FAIL);
  SETTIMER:ABSOLUTE (FAILURE, 500);
END;

/CONTROL/ TMAX = 1000;
          OPTION = NRESULT;

/EXEC/ BEGIN
         SETEXCEPT:HANDLER (SIMSTART, INITIMER);
         SIMUL;
         PRINT ("Maximum number of items:",
                GETSTAT:CUSTNB:MAXIMUM (PACKAGE));
       END;
Maximum number of items:   7.000
/END/
```

In this example, the timer object FAILURE is used to model a deterministic failure of a packaging machine. The failure date is scheduled with SETTIMER:ABSOLUTE. The failure is modeled by blocking the queue PACKAGE. The queue is restarted after a deterministic repair time, programmed with SETTIMER:RELATIVE.

The timer is initialized within the SIMSTART exception, because SETTIMER:*keyword* procedures are not allowed outside simulation.

Solvers $\boxed{5}$

5.1 Introduction

5.1.1 Overview

In QNAP2, solving a model by means of a solver consists in computing or estimating the performance criteria which characterize the model.

The mathematical solvers available in the current QNAP2 version are activated by the following procedures:

- **SOLVE procedure:** analytical solvers (exact and approximate)
- **MARKOV procedure:** markovian solver (exact)
- **SIMUL procedure:** event-driven simulation (approximate)

The exact solvers produce the exact theoritical values of the performance criteria characterizing the model. The approximate solvers produce an estimate of the exact solution, but this approximation is generally sufficient in practical cases.

Studying a model by means of an event-driven simulation (SIMUL procedure) consists in reproducing the behaviour of the different model components (stations and customers) or more precisely, one sample of the possible behaviours of the model components in the case of a stochastic model. With some caution a simulation can produce a good estimate of the model performance criteria and can therefore be regarded as an approximate solver.

5.1.2 Bringing resolution into play

Before calling a solution method the user may select a sub-model by using the NETWORK procedure which defines the sub-network on which the solution method is applied. The parameters of the stations belonging to the sub-network should reference only stations within the same sub-network:

Example:

```
/DECLARE/ QUEUE A, B, C;

/STATION/ NAME    = A;
          TRANSIT = C;

/EXEC/    BEGIN           & this statement
            NETWORK(A,B); & is incorrect, because A
            SOLVE;        & references C which is
          END;            & outside the NETWORK (A,B)
```

The user may also define an executable entry block (ENTRY parameter) and an executable exit block (EXIT parameter) to be executed just before and just after the resolution procedure is invoked.

The solvers of QNAP2 may then be activated by using one of the resolution procedures:

- **SOLVE:** for analytical solvers,
- **MARKOV:** for the Markovian solver,
- **SIMUL:** for the discrete event simulation.

The system then enters initiation time during which the evaluation of the following parameters of each station of the network is done. This evaluation is performed once only (these parameters are not re-evaluated during solution time).

- probabilistic transitions (parameters TRANSIT, SPLIT, FISSION),
- customer merges (parameters MATCH, FUSION),
- initial number of customers (parameter INIT),
- number of servers (parameter TYPE),
- queue capacity (parameter CAPACITY)
- priority levels (parameter PRIOR),
- quantum values (parameter QUANTUM),
- service rates (parameter RATE).

When initiation time is completed, the system enters solution time.

1. if SOLVE was activated, the SERVICE parameters are then evaluated in order to get the work demands distributions and the REJECT parameters are evaluated to get the rejection policy (these evaluations are performed once only) and the analysis proceeds.
2. if MARKOV or SIMUL was activated, the statements associated with the SERVICE and REJECT parameters are then dynamically executed during the resolution process.

Intermediate results produced by a resolution procedure may be obtained by setting the TRACE option of the OPTION parameter of the /CONTROL/ command. The data produced by the TRACE option is specific to each solver.

5.1.3 Results

After the normal completion of a resolution procedure, the performance criteria characterizing the steady state of the model are printed in a standard report.

This report is not printed if the NRESULT option has been specified (see the OPTION parameter of the /CONTROL/ command).

Additional results may be requested using /CONTROL/ command parameters or SET-STAT:*keyword* procedures. The computable results depend on the solver used:

- Class-specific results can be computed by all solvers except some analytical solvers.
- Marginal probabilities on the number of customers in the stations can be computed by all solvers except some analytical solvers.
- Confidence intervals may be computed by the discrete-event simulator in order to validate the steady-state assumption.

The computed values may also be accessed in the model itself by means of the built-in result access functions (see section 5.5.3.2 on page 234).

The performance criteria printed in the standard report are:

mean utilization factor: the utilization factor of a station is the percentage of busy servers (or busy resource units) in the station. It is equal to zero in the case of an infinite station.

mean service time: this is the service time as seen by the customers.

mean response time: the response time is the sum of the waiting time and service time.

mean number of customers: this is the number of customers being served or waiting.

mean throughput: defined as the mean number of customer services completed per time unit.

For the exact definition of the values obtained by a simulation see section 5.5 “Simulation results”, on page 226.

Example:

```
/DECLARE/ QUEUE S, A, B;
          CLASS X,Y;

/STATION/ NAME    = S;
          TYPE    = SOURCE;
          TRANSIT = A,X,1,A,Y,1;
          SERVICE = EXP(10.);

/STATION/ NAME    = A;
          TRANSIT = B,X,1,OUT,3;
          SERVICE = EXP(1.);

/STATION/ NAME    = B;
          TRANSIT = A;
          SERVICE = EXP(3.);

/CONTROL/ CLASS    = A;
          MARGINAL = B, 3;

/EXEC/    SOLVE;

/END/
```

Standard Report:

```
- CONVOLUTION METHOD ("CONVOL") -
*******************************************************************
* NAME     * SERVICE  * BUSY PCT * CUST NB  * RESPONSE * THRUPUT  *
*******************************************************************
*          *          *          *          *          *          *
* S        * 10.00    * 1.000    * 1.000    * 10.00    * .1000    *
*          *          *          *          *          *          *
* A        * 1.000    * .1333    * .1538    * 1.154    * .1333    *
*(X)       * 1.000    *0.8333E-01*0.9615E-01* 1.154    *0.8333E-01*
*(Y)       * 1.000    *0.5000E-01*0.5769E-01* 1.154    *0.5000E-01*
*          *          *          *          *          *          *
* B        * 3.000    * .1000    * .1111    * 3.333    *0.3333E-01*
*          *          *          *          *          *          *
*******************************************************************

*** MARGINAL PROBABILITIES : STATION B        ***

GLOBAL         : CUSTNB=     0     1     2     3
                 PROB= 0.900 0.090 0.009 0.001
                 VARIANCE OF CUSTOMER NUMBER =        0.123

             MEMORY USED:       2757 WORDS OF 4 BYTES
             ( 5.51 % OF TOTAL MEMORY)
```

5.2 Analytical Solvers

5.2.1 Overview

QNAP2 includes a set of specialized analytical solvers which yield exact or approximate steady-state solutions. These solvers are activited by the SOLVE procedure. They apply only to models verifying some fundamental assumptions:

transitions: the transitions performed at the completion of a service may be class dependent, the class of a customer may be modified during a transition, and the transition probabilities may not depend on the state of the network.

services: a service is restricted to a single work demand procedure and may not include any other operation. The work demand may not depend on the state of the network. No synchronization procedure may be used.

The available distributions for specifying work demands are:

- EXP (m) : exponential (m = mean),
- HEXP (m, c2): hyper-exponential (m = mean, c2 = squared coefficient of variation),
- ERLANG(m, k): Erlang (m = mean, and k = steps (c2 = 1/k)),
- CST (m): constant m,
- COX (t): Coxian law with coefficients in table t.

rates: service rates may only depend on the current number of customers in the station, globally or per class.

station types: resources, semaphores, flags and simple queues without service specification are not allowed.

scheduling: quantum scheduling is not allowed.

In addition to these basic assumptions, there are some additional functional limitations described in the next section. Note that if the analytical solvers apply, the computation time is always smaller than with the Markovian solver or with the simulator (MARKOV and SIMUL procedures).

Since QNAP2 includes a large set of analytical solvers, an automatic dispatching facility is built in which searches for the most appropriate solver depending on the characteristics of the model. Nevertheless, the user may request the invocation of any specific solver (controlled dispatching mode).

5.2.2 Functional limitations

5.2.2.1 Definitions

This paragraph describes various terms used or implied throughout the description of the analytical solvers.

sub-chain: a sub-chain is a set of pairs (station, class) which communicate with one another according to the transitions described in the model.

For instance, the pair (station A, class X) communicates with the pair (station B, class Y) if a customer being in station A with class X may reach the station B with class Y after one or more transitions.

Sub-chains contain distinct customer populations moving from one station to another or changing from one class to another but which do not mix themselves. A sub-chain is said to be closed if its population is constant (fixed at initialization time). Such a sub-chain may not involve a source or exit station (OUT queue). A sub-chain is said to be open if it contains at least one source and one exit station.

closed network: a network is said to be closed if all its sub-chains are closed.

open network: a network is said to be open if all its sub-chains are open.

mixed network: a mixed network is made up of at least one closed sub-chain and at least one open sub-chain.

Example:

```
& the services are not described

/DECLARE/ QUEUE A, B, C;
          CLASS X, Y, Z;

/STATION/ NAME       = A;
          TRANSIT(X) = B, Z ;
          TRANSIT(Y) = B,0.2,OUT;

/STATION/ NAME       = B;
          TRANSIT    = A;
          TRANSIT(Z) = A,X;

/STATION/ NAME    = C;
          TYPE    = SOURCE;
          TRANSIT = B,Y,0.1,A,Y;
```

There are two sub-chains in this network:

```
{ (A,X) (B,Z) }                 : closed sub-chain,
{ (C,.) (A,Y) (B,Y) (OUT,.) }   : open sub-chain,
```

therefore the network is a mixed network.

5.2.2.2 Network topology

The implemented analytical solvers overimpose the following conditions concerning the network topology:

5.2.2.2.1 open sub-chain An open sub-chain should contain only one source station.

Example:
Incorrect network

```
/DECLARE/ QUEUE S1, S2, A;

/STATION/ NAME    = S1;
```

```
          TYPE    = SOURCE;
          TRANSIT = A;
          SERVICE = EXP (10);

/STATION/ NAME = S2;
          COPY = S1;

/STATION/ NAME    = A;
          TRANSIT = OUT;
          SERVICE = EXP(1);
```

The two sources S1 and S2 send customers of the same class in A. Two customer classes should be used in this case.

5.2.2.2.2 open network In an open network one should have:

- either only one source for all the sub-chains,
- either a different source for each sub-chain.

Example:
Incorrect network

```
/DECLARE/ QUEUE S1, S2, A, B, C;
          CLASS X, Y, Z;

/STATION/ NAME    = S1;
          TYPE    = SOURCE;
          TRANSIT = A,X, 1, B,Y, 1;
          SERVICE = EXP (10);

/STATION/ NAME    = S2;
          TYPE    = SOURCE;
          TRANSIT = C,Z;
          SERVICE = EXP (20);

/STATION/ NAME    = A;
          TRANSIT = OUT;
          SERVICE = EXP (1);

/STATION/ NAME = B;
          COPY = A;

/STATION/ NAME = C;
          COPY = A;
```

The source S1 sends customers in two sub-chains:

```
{ (S1,.) (A,X) (OUT,.) },
```

and

```
{ (S1,.) (B,Y) (OUT,.) }.
```

The source S2 sends customers in the sub-chain:

```
{ (S2,.) (C,Z) (OUT,.) }.
```

One should have either three distinct sources or a single source.

5.2.2.2.3 mixed network In a mixed network one should have a different source for each open sub-chain.

Example:
Incorrect network

```
/DECLARE/ QUEUE S, A;
          CLASS X, Y, Z;

/STATION/ NAME    = S;
          TYPE    = SOURCE;
          TRANSIT = A, X, 1, A, Y, 1;
          SERVICE = EXP (10);

/STATION/ NAME        = A;
          SCHED       = PS;
          INIT (Z)    = 5;
          TRANSIT (X) = OUT;
          TRANSIT (Y) = OUT;
          TRANSIT (Z) = A;
          SERVICE (X) = EXP (1);
          SERVICE (Y) = EXP (2);
          SERVICE (Z) = EXP (3);
```

There are two open sub-chains:

```
{ (S,.) (A,X) (OUT,.) }
{ (S,.) (A,Y) (OUT,.) }
```

and one closed sub-chain:

```
{ (A,Z) }
```

The two open sub-chains include the same source station.

5.2.2.2.4 source station A source station may not send customers directly to the OUT queue.

Example:
Incorrect network

```
/DECLARE/ QUEUE S, A;

/STATION/ NAME    = S;
          TYPE    = SOURCE;
          TRANSIT = A, 0.96, OUT;
```

5.2.2.2.5 customer class

A given customer class must belong to only one sub-chain.

Example:

Incorrect network

```
/DECLARE/ QUEUE A, B, C;
          CLASS X,Y;

/STATION/ NAME       = A;
          TRANSIT(X) = B;
          TRANSIT(Y) = C;

/STATION/ NAME    = B;
          TRANSIT = A;

/STATION/ NAME       = C;
          TRANSIT(X) = B,Y;
          TRANSIT(Y) = C,X;
```

There are two sub-chains:

```
{ (A,X) (B,X) }
{ (A,Y) (C,Y) (C,X) (B,Y) }
```

The two sub-chains are distinct but class X belongs to both of them.

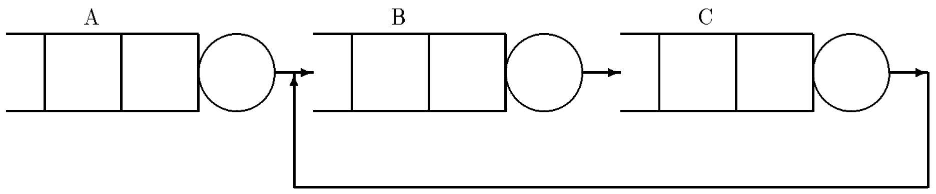
5.2.2.3 Ergodicity conditions

The analysis can be performed only if all the sub-chains are ergodic. A sub-chain is said to be ergodic if for any pair (station A, class X) a customer of the sub-chain leaving station A with class X will return to station A with class X after a finite period.

The ergodicity conditions are verified in QNAP2 taking into account the initial state of the network. The transient states, i.e. the pairs (station, class) which contain customers in the initial state of the network but which do not contain customers in the steady state, are flagged by a WARNING.

If a non ergodic sub-chain is detected, the associated transitions are suppressed from the network and the analysis process continues.

Example:



```
/DECLARE/ QUEUE A, B, C;

/STATION/ NAME    = A;
          INIT    = 1;
          TRANSIT = B;
          SERVICE = EXP (1);
```

```
/STATION/ NAME    = B;
          TRANSIT = C;
          SERVICE = EXP (1);

/STATION/ NAME    = C;
          TRANSIT = B;
          SERVICE = EXP (1);
```

Sub-chain A B C is not ergodic since a customer leaving station A will never return to this station.

5.2.3 Dispatching principles

The dispatcher works on an ordered chain of solvers built from a subset of the available solvers. This subset is defined according to the following rules:

5.2.3.1 Automatic dispatching

The subset of solvers used by the dispatcher is then the whole set of currently implemented solvers in QNAP2. These solvers are ordered according to the following rules:

exact solvers	$>$	approximate solvers
numerically safe solvers	$>$	numerically unsafe solvers
fast solvers	$>$	slow solvers

5.2.3.2 Controlled dispatching

The subset of solvers used by the dispatcher along with their order are specified by the user in the parameter list of the procedure SOLVE.

Example:

```
SOLVE ( "solver1, "solver2", "solver3" ) ;
```

The set of solvers is restricted to solver1, solver2 and solver3 ordered as specified in the parameter list of SOLVE.

5.2.3.3 Dispatching algorithm

```
current_solver <-- first_solver
repeat
    repeat
        select current_solver
        check if current_solver applies to the model
        if match exit
        else when current_solver = last_solver raise error
        current_solver <-- next_solver
    run selected solver
    if successful exit
    else when current_solver = last_solver raise error
    current_solver <-- next_solver
```

5.2.4 Tracing facilities

When the TRACE option is set (OPTION parameter of the CONTROL command), a set of standard intermediate results is produced by the solver. Besides, specific additionnal results are produced by each solver. These specific outputs are described in the sections devoted to each solver.
The standard intermediate results consist of the following data:

- list of the non-zero transition probabilities,
- results of sub-chain analysis: number and type of subchains, arrival rates in each station, number of iterations for the computation of the arrival rates,
- number of customers in each class and in each subchain,
- mean and squared coefficient of service times for each station and each class.

Example:

```
/DECLARE/ QUEUE S, A, B, C;
          INTEGER N;
          CLASS X,Y;

/STATION/ NAME    = S;
          TYPE    = SOURCE;
          TRANSIT = A,X,1,A,Y,1;
          SERVICE = EXP(10.);

/STATION/ NAME    = A;
          TRANSIT = B,X,1,C,Y,2,OUT,3;
          SERVICE = EXP(1.);

/STATION/ NAME    = B,C;
          TRANSIT = A;
          SERVICE = EXP(3.);

/CONTROL/ OPTION  = TRACE;

/EXEC/ SOLVE;
```

Standard intermediate results:

```
TRANSITIONS
***********
(S        ,X        ) TO (A        ,X        ) 0.50000
                         (A        ,Y        ) 0.50000
(S        ,Y        ) TO (A        ,X        ) 0.50000
                         (A        ,Y        ) 0.50000
(A        ,X        ) TO (B        ,X        ) 0.16667
                         (C        ,Y        ) 0.33333
                         OUT                    0.50000
(A        ,Y        ) TO (B        ,X        ) 0.16667
                         (C        ,Y        ) 0.33333
                         OUT                    0.50000
```

```
(B       ,X       ) TO (A       ,X       ) 1.00000
(C       ,Y       ) TO (A       ,Y       ) 1.00000

SUBCHAIN ANALYSIS
*****************
*** NETWORK IS OPEN
CLASS:X  BELONGS TO THE OPEN SUBCHAIN: 1 THE SOURCE OF WHICH IS:S
CLASS:Y  BELONGS TO THE OPEN SUBCHAIN: 1 THE SOURCE OF WHICH IS:S

ARRIVAL RATES AFTER       0 ITERATIONS ... ERROR : 0.100E+01
STATION     S           A           B           C
X          1.000       1.000       1.000      0.0000E+00
Y          1.000       1.000      0.0000E+00 1.000

ARRIVAL RATES AFTER      19 ITERATIONS ... ERROR : 0.781E-05
STATION     S           A           B           C
X         0.0000E+00 .8333       .3333      0.0000E+00
Y         0.0000E+00 1.167      0.0000E+00 .6667

-- CV2 --
*********
STATION     S           A           B           C
X          1.000       1.000       1.000       1.000
Y          1.000       1.000       1.000       1.000

MEAN SERVICE TIMES
******************
STATION     S           A           B           C
X          10.00       1.000       3.000       3.000
Y          10.00       1.000       3.000       3.000
```

the intermediate results specific to the solver selected by the dispatcher and the standard report are not given here.

5.2.5 List of the analytical solvers

The following solvers are implemented in QNAP2:

CONVOL (convolution algorithms): CONVOL is a fast, exact, but numerically unsafe solver. It implements the product-form theorems of Baskett, Chandy, Muntz and Palacios with convolution algorithms.

MVA (mean value analysis): MVA is a fast, exact and numerically safe solver. It implements the mean value analysis approach developed by M. Reiser. Its conditions of application are more restrictive than those of CONVOL.

MVANCA (mean value analysis and normalized convolution algorithms): MVANCA is a fast, exact and numerically safe solver. It implements an hybrid method based on the mean value analysis and the normalized convolution algorithms proposed by M. Reiser. Its conditions of applications are more restrictive than those of CONVOL.

HEURSNC (Schweitzer, Neuse and Chandy heuristic algorithms): HEURSNC is a fast, approximate and numerically unsafe solver. It implements a heuristic proposed by Schweitzer, Neuse and Chandy. Its conditions of application are more restrictive than those of CONVOL.

PRIORPR (Veran algorithm for preemptive priority scheduling): PRIORPR is a fast, approximate and numerically safe solver. It implements an algorithm developed by M. Veran for the solution of closed multiclass queueing networks with preemptive priority scheduling.

ITERATIV (Marie iterative algorithms): ITERATIV is a fast, approximate and numerically unsafe solver. It implements a heuristic developed by R. Marie. It applies to closed mono-class networks with non-exponential FIFO servers.

DIFFU (Gelenbe and Pujolle diffusion algorithms): DIFFU is a fast, approximate and numerically safe solver. It implements a heuristic developed by E. Gelenbe and G. Pujolle. It applies to open multi-class networks with non-exponential FIFO servers.

SPLITMAT (Baccelli *et al.* algorithm for parallel split-match networks): SPLITMAT is a fast, approximate and numerically safe solver. It implements a bounding algorithm developped by F. Baccelli *et al.* It applies to open, series-parallel networks with exponential FIFO single server stations.

5.2.6 The convolution solver CONVOL

5.2.6.1 Overview

CONVOL is the direct application of the product-form theorems due to Baskett, Chandy, Muntz and Palacios. These theorems extend to open, closed and mixed multiclass queueing networks previous results obtained by R. R. Jackson. The product form theorems have been also extended to networks with finite-capacity queues, and/or multiple server stations with concurrency classes of customers.

This solver uses efficient computing algorithms that can handle large networks with small computation times (especially as compared to simulation times), and that produce the exact steady state solution.

The main restriction of this method is to require exponentially distributed service times or scheduling disciplines leading to similar properties.

5.2.6.2 Bibliography

BASKETT, F., CHANDY, K.M., MUNTZ, R.R. and PALACIOS F.G. *Open, Closed and Mixed Networks of Queues with Different Classes of Customers.* J.ACM 22, 2, April 1975, pp. 248-260.
REISER, M. and KOBAYASHI, H. *Recursive algorithms for general queuing networks.* IBM J. of Res. and Dev. May 1975, pp. 283-294
MERLE, D. *Algorithmes de calcul des probabilités stationnaires d'un réseau de files d'attente.* Rapport de Recherche INRIA 279, mars 1978, 62 pages.
MERLE, D. *Contribution à l'étude d un analyseur de modèles à réseaux de files d'attente.* Thèse de Docteur-ingenieur, INPL, Nancy, oct. 1978.
LE BOUDEC, J.-Y. *A BCMP extension to Multiserver Stations with Concurrent Classes of Customers.* In Proceedings of Performance'86 and ACM Sigmetrics, Performance Evaluation Review, Special Issue Vol. 14, No. 1, May 1986, pp 78-91.
MUSSI, Ph. and NAIN, Ph. *Description and Specifications for New Product-Form Queueing Network Stations.* ESPRIT Project 2143 IMSE report R 5.5-3 V1, March 1989.

5.2.6.3 Application stipulations

network topology: The network may be open, closed or mixed with some topological constraints (see below).

customer classes: Several customer classes may be used (having different service requirements and different routing rules).

station types: The following station types are allowed:

- simple server,
- multiple server,
- infinite server,
- source.

Resources, semaphores, flags and simple queues (stations without service) may not be used.

Note:

If the number of servers of a multiple station is always greater than or equal to the number of customers in the network this station is treated as an infinite station.

scheduling: The following scheduling procedures are allowed:

- FIFO,
- LIFO, PREEMPT,
- PS,
- FEFS, EXCLUDE.

Priorities and quantum allocation may not be used.

When concurrency sets are used (EXCLUDE scheduling),

- the definition of the concurrency sets should be performed either with the classes or with probabilities only. Definitions mixing classes and probabilities is not allowed.
- *all* customers must belong to a concurrency set (no "outsiders"), and
- the FEFS option is implied.

Example:

```
/DECLARE/ CLASS C1, C2, C3, C4, C5;
...

SCHED = FEFS, EXCLUDE (C1, C2), (C3, C4, C5);      & ok

SCHED = FEFS, EXCLUDE (0.4, 0.4, 0.2);             & ok

SCHED = FEFS, EXCLUDE (C1, 0.4, C2, 0.7),          & mixing
                      (C3, 0.6, C4, 0.3),          & prob/class
                      (C5);                        & not allowed

SCHED = FEFS, EXCLUDE (C1, C2),                    & C5 missing
                      (C3, C4);                    & not allowed
```

Note:

If concurrency sets are used, then the following restrictions apply to the station:

1. The capacity must be infinite.
2. The service time distribution must be exponential.

station capacity: The station capacity may be infinite (default) or finite.

If finite capacity stations are used, the capacity limitation may not depend on classes: only a global capacity is allowed.

The reject sequence (REJECT parameter) is limited to the SKIP procedure.

A finite capacity queue may not use concurrency sets.

Note:

1. If the network is closed and contains only finite capacity stations, the customer population must be lower than the sum of station capacities.
2. If the network is mixed, the previous statement applies to any closed sub-chain.

service: Each service has to be limited to a single work demand with either:

- an exponential distribution, identical for all the customer classes if the scheduling is FIFO or FEFS, EXCLUDE.
- a general distribution, which may be distinct for each customer class if the scheduling is PS or LIFO, PREEMPT or if the station is an infinite server station (TYPE = INFINITE).

In fact, one can prove that for the stations accepting general service time distributions, only the mean of the service distribution is significant. All the available distributions lead to the same results if they have the same mean value.

A service must be limited to a single work demand procedure and may not include any other operation. Moreover, the mean service time may not be null. The work demands may not depend on the network state.

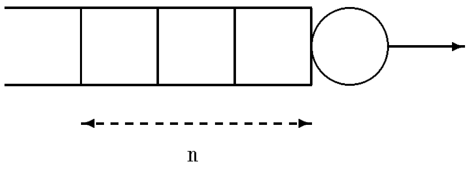
No synchronization procedure (P, V, WAIT,...) may be used.

transition rules: The transition performed at service completion time may be class dependent.

- The class of a customer may be modified during a transition.
- Transition probabilities may not depend on the state of the network.
- A transition probability may be null (no transition to the corresponding station).

service rate: The service rate may be different for each customer class. It has to be a constant service rate except in the following cases where it may depend on the number of customers in the station:

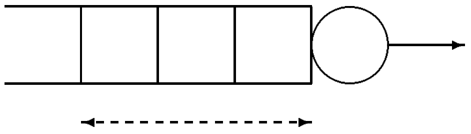
- for a single server FIFO station the service rate may depend on the total number of customers in the station.



RATE $= a_1, a_2, \ldots a_n$

The service rate is equal to a_i if the station contains exactly i customers (or a_n for $i > n$)

- for a single server station with PS or LIFO, PREEMPT discipline the service rate of a given customer class may depend on the total number of customers in the station and/or on the number of customers of this class.



$n_1, \ldots, n_r, \ldots, n_k$

RATE $= a_1, a_2, \ldots a_n$

RATE $(C_r) = a_{r_1}, a_{r_2}, \ldots a_{r_n}$

Class C_r customers service rate is: $a_i * a_{r_j}$

The class dependencies may be used only in a closed network.

5.2.6.4 Application stipulations summary

station type	scheduling	service time distribution	distinct serv. time per class	global dependent serv. rate	dependent serv. rate per class
SINGLE	FIFO	exponential	no	yes	no
	LIFO, PREEMPT	general	yes	yes	yes
	PS	general	yes	yes	yes
MULTIPLE	FIFO, EXCLUDE	exponential	no	yes	no
	PS	general	yes	yes	yes
INFINITE	not relevant	general	yes	yes	yes
SOURCE	not relevant	exponential	no	no	no

Example:

In this example a queueing network consisting of two stations A and B is described and analysed by the solver CONVOL with various specifications of stations A and B.

```
/DECLARE/ QUEUE A,B;
          CLASS X, Y;
          REAL T = 2., TX = 1.5, TY = 0.5;
          REAL R(2) = (2,3), RX(2) = (3,4), RY(2) = (4,3);
          INTEGER N = 3;

/STATION/ NAME    = B;
          TYPE    = INFINITE;
          INIT    = N;
          TRANSIT = A;
          SERVICE = CST(1.);

/STATION/ NAME    = A;
          RATE    = R;
          TRANSIT = B;
          SERVICE = EXP(T);

/EXEC/    SOLVE("CONVOL");

/STATION/ NAME = A;
          TYPE = MULTIPLE (3);

/EXEC/    SOLVE("CONVOL");

/STATION/ NAME       = A;
          TYPE       = SINGLE;
          SCHED      = LIFO, PREEEMPT;
          RATE(X)    = RX;
          RATE(Y)    = RY;
          SERVICE(X) = CST(TX);
          SERVICE(Y) = CST(TY);

/EXEC/    SOLVE("CONVOL");
```

```
/STATION/ NAME  = A;
          SCHED = PS;

/EXEC/    SOLVE("CONVOL");

/STATION/ NAME = A;
          TYPE = MULTIPLE (3);
          SCHED = PS;

/EXEC/    SOLVE("CONVOL");

/STATION/ NAME = A;
          TYPE = INFINITE;

/EXEC/    SOLVE("CONVOL");

/STATION/ NAME    = B;
          TRANSIT = OUT;

/STATION/ NAME    = A;
          TYPE    = SOURCE;
          SERVICE = EXP(T);
          TRANSIT = B,X,1,B,Y,1;

/EXEC/    SOLVE("CONVOL");

/STATION/ NAME = B;
          TYPE = SINGLE;
          SCHED = FIFO;
          CAPACITY = 5;
          REJECT = SKIP;
          SERVICE = EXP (1);

/EXEC/    SOLVE("CONVOL");

/STATION/ NAME = B;
          TYPE = MULTIPLE (3);
          SCHED = FEFS,EXCLUDE (X), (Y);
          CAPACITY = ;

/EXEC/    SOLVE("CONVOL");


/END/
```

5.2.6.5 Tracing facilities

When the TRACE option is set, the standard intermediate results of SOLVE and results about the normalization constants are produced.

5.2.6.6 Marginal probabilities

Marginal probabilities can be computed with CONVOL solver excepted if an EXCLUDE discipline has been defined.

5.2.7 The Mean Value Analysis solver MVA

5.2.7.1 Overview

MVA was developed in order to avoid the computation of the normalizing constants which may lead to numerical unstabilities. MVA yields exact results.

5.2.7.2 Bibliography

SEVCIK, K.C. and MITRANI, I. *The distribution of queueing network states at input and output instant.* Rapport de recherche INRIA 307, mai 1978.
REISER, M. and LAVENBERG, S. *Mean-Value Analysis of Closed Multichain Queuing Networks.* J.ACM 27,2, April 1980, pp. 313-322.
DRIX, P. *Etude d'algorithmes de résolution de réseaux de files d'attente fermés BCMP et heuristiques associées.* Thèse de docteur-ingénieur, Université de Paris VI, nov. 1982.

5.2.7.3 Application stipulations

Application stipulations of the solver MVA are similar to those of the solver CONVOL with the following additions:

- closed networks,
- no dependent service rates,
- no concurrency sets,
- no limited capacity stations,
- marginal probabilities may not be requested,
- single server or infinite server.

5.2.7.4 Application stipulations summary

station type	scheduling	service time distribution	distinct serv. time per class	global dependent serv. rate	dependent serv. rate per class
SINGLE	FIFO	exponential	no	no	no
	LIFO,PREEMPT	general	yes	no	no
	PS	general	yes	no	no
INFINITE	not relevant	general	yes	no	no
SOURCE	not relevant	exponential	no	no	no

Example:
In this example a queueing network consisting of two stations A and B is described and analysed by the solver MVA with various specifications of stations A and B.

```
/DECLARE/ QUEUE A,B;
          CLASS X, Y;
          REAL T = 2., TX = 1.5, TY = 0.5;
          REAL R(2) = (2,3), RX(2) = (3,4), RY(2) = (4,3);
          INTEGER N = 3;

/STATION/ NAME    = B;
          TYPE    = INFINITE;
```

```
                     INIT    = N;
                     TRANSIT = A;
                     SERVICE = CST(1.);

           /STATION/ NAME    = A;
                     TRANSIT = B;
                     SERVICE = EXP(T);

           /EXEC/    SOLVE("MVA");

           /STATION/ NAME       = A;
                     TYPE       = SINGLE;
                     SCHED      = LIFO, PREEEMPT;
                     SERVICE(X) = CST(TX);
                     SERVICE(Y) = CST(TY);

           /EXEC/    SOLVE("MVA");

           /STATION/ NAME  = A;
                     SCHED = PS;

           /EXEC/    SOLVE("MVA");

           /STATION/ NAME = A;
                     TYPE = INFINITE;

           /EXEC/    SOLVE("MVA");
```

5.2.7.5 Tracing facilities

When the TRACE option is set, only the standard intermediate results of SOLVE are produced (no specific intermediate results).

5.2.8 The Mean Value Analysis, Normalized Convolution Algorithm MVANCA

5.2.8.1 Overview

MVA and NCA were developed in order to avoid the computation of the normalizing constants which may lead to numerical instabilities. MVANCA yields exact results.

5.2.8.2 Bibliography

REISER, M. *Mean-Value Analysis and Convolution Method for Queue-Dependent Servers in Closed queuing Networks.* Performance Evaluation Review 1, 1, Jan. 1981
DRIX, P. *Etude d'algorithmes de résolution de réseaux de files d'attente fermés BCMP et heuristiques associées.* Thèse de docteur-ingénieur, Université de Paris VI, nov. 1982.
MUSSI, Ph. and NAIN, Ph. *Description and Specifications for New Product-Form Queueing Network Stations.* ESPRIT Project 2143 IMSE report R 5.5-3 V1, March 1989.

5.2.8.3 Application stipulations

The application stipulations are similar to those of the solver CONVOL with the following additions:

- closed networks,
- mono-chain networks,
- no dependent service rates per class,
- marginal probabilities may not be requested,
- no concurrency sets.

5.2.8.4 Application stipulations summary

station type	scheduling	service time distribution	distinct serv. time per class	global dependent serv. rate	dependent serv. rate per class
SINGLE	FIFO	exponential	no	yes	no
	LIFO,PREEMPT	general	yes	yes	no
	PS	general	yes	yes	no
MULTIPLE	FIFO	exponential	no	yes	no
	PS	general	yes	yes	no
INFINITE	not relevant	general	yes	yes	no

Example:
In this example a queueing network consisting of two stations A and B is described and analysed by the solver MVANCA with various specifications of stations A and B.

```
/DECLARE/ QUEUE A,B; CLASS X, Y;
          REAL T = 2., TX = 1.5, TY = 0.5;
          REAL R(2) = (2,3), RX(2) = (3,4), RY(2) = (4,3);
          INTEGER N = 3;

/STATION/ NAME       = B;
          TYPE       = INFINITE;
          INIT       = N;
          TRANSIT(X) = A, Y;
          TRANSIT(Y) = A, X;
          SERVICE    = CST(1.);

/STATION/ NAME       = A;
          SERVICE    = EXP(T);
          RATE       = R;
          TRANSIT    = B;

/EXEC/    SOLVE("MVANCA");


/STATION/ NAME       = A;
          TYPE       = MULTIPLE (3);

/EXEC/    SOLVE("MVANCA");

/STATION/ NAME       = A;
          TYPE       = SINGLE;
          SCHED      = LIFO, PREEEMPT;
          SERVICE(X) = CST(TX);
          SERVICE(Y) = CST(TY);

/EXEC/    SOLVE("MVANCA");

/STATION/ NAME       = A;
          SCHED      = PS;

/EXEC/    SOLVE("MVANCA");

/STATION/ NAME       = A;
          TYPE       = MULTIPLE (3);
          SCHED      = PS;

/EXEC/    SOLVE("MVANCA");

/STATION/ NAME       = A;
          TYPE       = INFINITE;

/EXEC/    SOLVE("MVANCA");

/STATION/ NAME = B;
          TYPE = SINGLE;
          SCHED = FIFO;
          CAPACITY = 5;
          REJECT = SKIP;
          SERVICE = EXP (1);

/EXEC/    SOLVE("MVANCA");
```

```
&              /STATION/ NAME = B;
&                        TYPE = MULTIPLE (3);
&                        SERVICE = EXP (1);
&                        SCHED = EXCLUDE (X), (Y);
&                        CAPACITY = ;
&
&              /EXEC/    SOLVE("MVANCA");


             /END/
```

5.2.8.5 Tracing facilities

When the TRACE option is set, only the standard intermediate results of SOLVE are produced (no specific intermediate results).

5.2.9 The Heuristic solver HEURSNC

5.2.9.1 Overview

The solver HEURSNC was developed in order to solve networks with a large number of sub-chains. The method yields approximate results.

5.2.9.2 Bibliography

NEUSE, D. and CHANDY, K. *SCAT: A heuristic algorithm for queuing networks of computing systems.* ACM Sigmetrics 10, 1, Fall 1981, pp. 59-79.

5.2.9.3 Application stipulations

The application stipulations of HEURSNC are similar to those of the solver CONVOL with the following additions:

- closed network,
- no concurrency sets,
- no limited capacity stations,
- no dependent service rates per class,
- if R(i) is the i-th service rate coefficient, then R(i) must be a concave monotonically non-decreasing function of i,
- marginal probabilities may not be requested.

5.2.9.4 Application stipulations summary

station type	scheduling	service time distribution	distinct serv. time per class	global dependent serv. rate	dependent serv. rate per class
SINGLE	FIFO	exponential	no	yes	no
	LIFO,PREEMPT	general	yes	yes	no
	PS	general	yes	yes	no
MULTIPLE	FIFO	exponential	no	yes	no
	PS	general	yes	yes	no
INFINITE	not relevant	general	yes	yes	no

Example:

In this example a queueing network consisting of two stations A and B is described and analysed by the solver HEURSNC with various specifications of stations A and B.

```
/DECLARE/ QUEUE A,B; CLASS X, Y;
          REAL T = 2., TX = 1.5, TY = 0.5;
          REAL R(2) = (2,3), RX(2) = (3,4), RY(2) = (4,3);
          INTEGER N = 3;

/STATION/ NAME    = B;
          TYPE    = INFINITE;
          INIT    = N;
          TRANSIT = A;
          SERVICE = CST(1.);
```

```
/STATION/ NAME    = A;
          RATE    = R;
          TRANSIT = B;
          SERVICE = EXP(T);

/EXEC/    SOLVE("HEURSNC");

/STATION/ NAME    = A;
          TYPE    = MULTIPLE (3);

/EXEC/    SOLVE("HEURSNC");

/STATION/ NAME       = A;
          TYPE       = SINGLE;
          SCHED      = LIFO, PREEEMPT;
          SERVICE(X) = CST(TX);
          SERVICE(Y) = CST(TY);

/EXEC/    SOLVE("HEURSNC");

/STATION/ NAME  = A;
          SCHED = PS;

/EXEC/    SOLVE("HEURSNC");

/STATION/ NAME  = A;
          TYPE  = MULTIPLE (3);
          SCHED = PS;

/EXEC/    SOLVE("HEURSNC");

/STATION/ NAME = A;
          TYPE = INFINITE;

/EXEC/    SOLVE("HEURSNC");
```

5.2.9.5 Tracing facilities

When the TRACE option is set, only the standard intermediate results of SOLVE are produced (no specific intermediate results).

5.2.10 The Preemptive Priority solver PRIORPR

5.2.10.1 Overview

PRIORPR was developed in order to solve closed multiclass queueing networks involving premptive priority scheduling. It is an approximate method.

5.2.10.2 Bibliography

VERAN, M. *Exact analysis of a priority queue with finite source. Modeling and Performance Evaluation Methodology (F. Bacelli and G. Fayolle, eds.)*, Lectures Notes in Control and Information Sciences, Springer-Verlag, 60, 1984, pp. 371-390.

5.2.10.3 Application stipulations

The application stipulations of the solver PRIORPR are those of the solver CONVOL with the following additions:

- closed network,
- no class transitions (i.e. number of subchains = number of classes),
- one and only one single server station with PRIOR, PREEMPT scheduling, exponential service times, and distinct priority level for each class,
- results per class requested for the preemptive prior station (name of this station in the list of the CLASS parameter),
- no concurrency sets,
- no limited capacity stations,
- no dependent service rates,
- marginal probabilities may not be requested.

5.2.10.4 Application stipulations summary

station type	scheduling	service time distribution	distinct serv. time per class	global dependent serv. rate	dependent serv. rate per class
SINGLE	FIFO	exponential	no	no	no
	PRIOR,PREEMPT	exponential	yes	no	no
	LIFO,PREEMPT	general	yes	no	no
	PS	general	yes	no	no
INFINITE	not relevant	general	yes	no	no

Example:

In this example a queueing network consisting of two stations A and B is described and analysed by the solver PRIORPR with various specifications of stations A and B.

```
/DECLARE/ QUEUE A,B; CLASS X, Y;
          REAL T = 2., TX = 1.5, TY = 0.5;
          REAL R(2) = (2,3), RX(2) = (3,4), RY(2) = (4,3);
          INTEGER N = 3;

/STATION/ NAME       = B;
```

```
                    SCHED      = PRIOR, PREEMPT;
                    PRIOR(X)   = 1;
                    PRIOR(Y)   = 2;
                    INIT       = N;
                    TRANSIT    = A;
                    SERVICE(X) = EXP(TX);
                    SERVICE(Y) = EXP(TY);

          /CONTROL/ CLASS = B;

          /STATION/ NAME    = A;
                    TRANSIT = B;
                    SERVICE = EXP(T);

          /EXEC/    SOLVE("PRIORPR");

          /STATION/ NAME       = A;
                    TYPE       = SINGLE;
                    SCHED      = LIFO, PREEEMPT;
                    SERVICE(X) = CST(TX);
                    SERVICE(Y) = CST(TY);

          /EXEC/    SOLVE("PRIORPR");

          /STATION/ NAME      = A;
                    SCHED     = PS;

          /EXEC/    SOLVE("PRIORPR");

          /STATION/ NAME       = A;
                    TYPE       = INFINITE;

          /EXEC/    SOLVE("PRIORPR");
```

5.2.10.5 Control of numerical convergence

The convergence of the solver may be controlled by the two first parameters of the parameter CONVERGENCE of the CONTROL command:

- parameter 1: maximum number of iterations,
- parameter 2: threshold of the convergence test.

5.2.10.6 Tracing facilities

When the TRACE option is set, only the standard intermediate results of SOLVE are produced (no specific intermediate results).

5.2.11 The Iterative solver ITERATIV

5.2.11.1 Overview

ITERATIV was developed in order to solve closed queueing networks including stations having a FIFO discipline and a non-exponentially distributed service time. Therefore this solver implements one of the main restriction of the solver CONVOL.

ITERATIV is an approximate solver: it does not, in general, give the exact solution of the network (but it does produce exact results if all the service times are exponential).

The basic principle of this method is to replace the network with the non-exponential service times by an equivalent network with exponential service times. The criteria used to build this equivalent model are the conservation of the total number of customers in the model and the conservation of the throughputs in the network.

5.2.11.2 Bibliography

MARIE, R. *Méthodes itératives de résolution de modèles mathématiques de systèmes informatiques.* RAIRO Informatique 12, 2 1978.

5.2.11.3 Application stipulations

The application stipulations of this solver are identical to those of the solver CONVOL with the following additions:

- closed network,
- mono-class network,
- no concurrency sets,
- no limited capacity stations,
- FIFO single server station with general service time distribution,

Note:
ITERATIV approximates constant distributions by Erlang-5 distributions with the same mean.

5.2.11.4 Application stipulations summary

station type	scheduling	service time distribution	distinct serv. time per class	global dependent serv. rate	dependent serv. rate per class
SINGLE	FIFO	exponential	no	yes	no
	FIFO	general	no	no	no
	LIFO,PREEMPT	general	no	yes	no
	PS	general	no	yes	no
MULTIPLE	FIFO	exponential	no	yes	no
	PS	general	no	yes	no
INFINITE	not relevant	general	no	yes	no

Example:
In this example a queueing network consisting of two stations A and B is described and analysed by the solver ITERATIV with various specifications of stations A and B.

```
/DECLARE/ QUEUE A,B;
          REAL T = 2., TX = 1.5, TY = 0.5;
          REAL R(2) = (2,3), RX(2) = (3,4), RY(2) = (4,3);
          INTEGER N = 3;

/STATION/ NAME    = B;
          TYPE    = INFINITE;
          INIT    = N;
          TRANSIT = A;
          SERVICE = CST(1.);

/STATION/ NAME    = A;
          RATE    = R;
          TRANSIT = B;
          SERVICE = EXP(T);

/EXEC/    SOLVE("ITERATIV");

/STATION/ NAME    = A;
          RATE    = 1.;
          SERVICE = CST(T);

/EXEC/    SOLVE("ITERATIV");

/STATION/ NAME    = A;
          TYPE    = MULTIPLE (3);
          RATE    = R;
          SERVICE = EXP(T);

/EXEC/    SOLVE("ITERATIV");

/STATION/ NAME    =A;
          TYPE    = SINGLE;
          SCHED   = LIFO, PREEEMPT;
          SERVICE = CST(T);

/EXEC/    SOLVE("ITERATIV");

/STATION/ NAME  = A;
          SCHED = PS;

/EXEC/    SOLVE("ITERATIV");

/STATION/ NAME  = A;
          TYPE  = MULTIPLE (3);
          SCHED = PS;

/EXEC/    SOLVE("ITERATIV");

/STATION/ NAME = A;
```

```
                    TYPE = INFINITE;

          /EXEC/    SOLVE("ITERATIV");
```

5.2.11.5 Control of numerical convergence

The convergence of the iterative process may be controlled by the two first parameters of the parameter CONVERGENCE of the CONTROL command:

- parameter 1: maximum number of iterations,
- parameter 2: threshold of the convergence test.

5.2.11.6 Tracing facilities

When the TRACE option is set, the standard intermediate results of SOLVE are produced, completed by specific intermediate results obtained at each iteration of the iterative method. These intermediate results consists of:

- the relative arrival rates in each station as a function of the number of customers in this station,
- the standard report obtained at the end of each iteration (it is to be noted that the results found in those reports are meaningless as long as the convergence of the iterative process has not been reached).

5.2.12 The Difusion approximation solver DIFFU

5.2.12.1 Overview

DIFFU was developed to solve open queueing networks with stations having non-exponential service times and FIFO scheduling. DIFFU is an approximate solver.

The basic principle of this method is to compute the arrival rates for each station in the network and to estimate the coefficient of variation of the inter-arrival times. Then, for each station, an approximate solution of the GI/G/1 type is computed using diffusion approximations.

5.2.12.2 Bibliography

GELENBE, E. and PUJOLLE, G. *A Diffusion Model for Multiple Class queuing Networks,* Rapport de Recherche IRIA 242, aout 1977, 12 pages.

5.2.12.3 Application stipulations

The application stipulations of the solver DIFFU are identical to those of the solver CONVOL with the following additions:

- open network,
- inter-arrival times with general distributions,
- FIFO single server station with general service time distribution,
- no dependent service rates,
- no concurrency sets,
- no limited capacity stations,
- marginal probabilities may not be requested.

5.2.12.4 Application stipulations summary

station type	scheduling	service time distribution	distinct serv. time per class	global dependent serv. rate	dependent serv. rate per class
SINGLE	FIFO	general	yes	no	no
SOURCE	not relevant	general	yes	no	no

Example:

In this example a queueing network consisting of two stations A and B is described and analysed by the solver DIFFU with various specifications of stations A and B.

```
/DECLARE/ QUEUE A,B; CLASS X, Y;
          REAL T = 2., TX = 1.5, TY = 0.5;
          REAL R(2) = (2,3), RX(2) = (3,4), RY(2) = (4,3);
          INTEGER N = 3;

/STATION/ NAME       = B;
          TRANSIT    = OUT;
          SERVICE(X) = CST(TX);
          SERVICE(Y) = CST(TY);
```

```
/STATION/ NAME       = A;
          TYPE       = SOURCE;
          TRANSIT    = B,X,1,B,Y,1;
          SERVICE(X) = CST(TX);
          SERVICE(Y) = CST(TY);

/EXEC/    SOLVE("DIFFU");
```

5.2.12.5 Tracing facilities

When the TRACE option is set, the standard intermediate results of SOLVE are produced, completed by the two first moments of the inter-arrival time and service time for each station of the network.

5.2.13 The Split-Match approximation solver SPLITMAT

5.2.13.1 Overview

SPLITMAT was developed to solve series-parallel networks of exponential servers with FIFO scheduling, exponential arrivals, and SPLIT/MATCH synchronizations. SPLITMAT is an approximate solver that computes heuristic estimates based on performance *bounds.*

The lower bounds are based on the statistics of the stationary response times in D/M/1 queues. The upper bounds are based on independent response times in M/M/1 queues. The heuristic estimates are obtained as linear combinations of the upper and lower bounds.

The estimated results are the utilization of the servers, the length of the queues and the response time of the stations. In addition, the solver computes the distribution of the response time of the network, i.e., the total network traversal time.

Note: Due to the specific nature of the results computed by SPLITMAT, the results table produced by this solver is different from the standard results table. For the same reason, the results are *not* accessible by the usual result access functions (`GETSTAT:`*keyword*).

5.2.13.2 Bibliography

F. BACCELLI, A.M. MAKOWSKI. *Synchronization in Queueing Networks.* Proc. IEEE. Vol. 77, No. 1, Jan 1989.

F. BACCELLI, M. BADEL, Z. LIU. *Interim Report on Parallel Fork-Join Network Solvers.* Esprit Project 2143 IMSE. Report D5.5-1. 1990.

F. BACCELLI, Z. LIU. *Survey of Comparison Methods and Algorithms for Parallel Fork Join Networks.* Esprit Project 2143 IMSE. Interim report R5.5-5. 1990.

F. BACCELLI, A. JEAN-MARIE, Z. LIU. *A New Solver for Series-Parallel Fork Join Networks.* Esprit Project 2143 IMSE. Interim report R5.5-10. 1991.

F. BACCELLI, A. JEAN-MARIE, Z. LIU, P. NAIN. *Interim Report on New Solvers for Series-Parallel Networks and Sensitivity Analysis of Product-Form Networks.* Esprit Project 2143 IMSE. Report D5.5-3. 1991.

5.2.13.3 Application stipulations

The application stipulations of the solver SPLITMAT are identical to those of CONVOL with the following additions:

- open, series-parallel network, no loops ("feed-forward" network),
- only one source station with exponential service time distribution,
- FIFO single server stations with exponential service time distribution,
- no dependent service rates,
- no class-dependent service time distributions
- probabilistic transitions are not allowed,
- no limited capacity stations,
- in SPLIT specifications, *only one "piece"* can be sent to each destination queue,
- no probabilistic SPLIT,
- no probabilistic or priority MATCH: only one join-up specification is allowed for each MATCH,
- class results and marginal probabilities may not be requested.

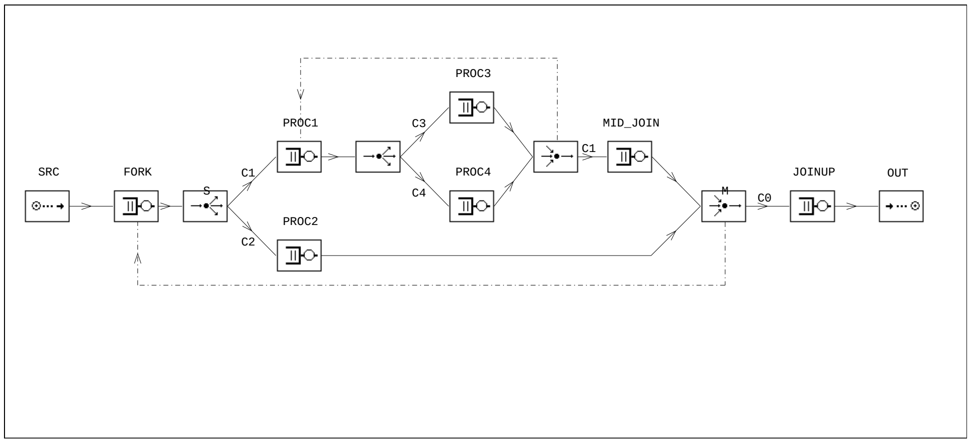


Figure 5.1: Series-parallel example. This queueing network corresponds to the second sample Qnap2 model ("dual stage split-match network").

The main restrictions are on the network topology: a series-parallel network starts at a single point (the SOURCE queue) and ends at a single point (the OUT queue). A customer exiting a station can be simply transited to the next queue (serial construct) or it can be split to a group of queues (parallel construct). A customer entering a queue can be joined-up with other customers. No loops are allowed: a customer may not be transited to a station where it has already been served. Figure 5.1 shows a typical series-parallel network.

5.2.13.4 Application stipulations summary

station type	scheduling	service time distribution	distinct serv. time per class	global dependent serv. rate	dependent serv. rate per class
SINGLE	FIFO	exponential	no	no	no
SOURCE	not relevant	exponential	not relevant	no	not relevant

Example:

In this example, a simple series-parallel queueing network with one SPLIT and one MATCH is analyzed with the SPLITMAT solver. A second example, with two pairs of SPLIT and MATCH parameters is also shown. Note the result table which is rather different from the standard result table, due to the specific nature of the computed results.

```
SIMULOG   ***  QNAP2  ***  ( 01-06-94  ) V 9.1
(C)  COPYRIGHT BY CII HONEYWELL BULL AND INRIA, 1986


& 1) Basic split-match network

/DECLARE/ QUEUE SRC, FORK, PROC1, PROC2, JOINUP, FINISH;
          CLASS C0, C1, C2, C3, C4;

/STATION/ NAME = SRC;
          TYPE = SOURCE;
          SERVICE = EXP (1.8);
          TRANSIT = FORK, C0;

/STATION/ NAME = FORK;
          SERVICE = EXP (0.9);
          SPLIT = (PROC1, C1, 1, PROC2, C2, 1);

/STATION/ NAME = PROC1;
          SERVICE = EXP (1.4);
          TRANSIT = JOINUP;

/STATION/ NAME = PROC2;
          SERVICE = EXP (0.01);
          TRANSIT = JOINUP;

/STATION/ NAME = JOINUP;
          SERVICE = EXP (1.0);
          MATCH = (FORK):(C1, 1, C2, 1) C0;
          TRANSIT = FINISH;

/STATION/ NAME = FINISH;
          SERVICE = EXP (1.2);
          TRANSIT = OUT;

/EXEC/ SOLVE ("SPLITMAT");
 - BOUNDS FOR SPLIT/MATCH SYSTEMS ("SPLITMAT") -

******************  RESULTS ON INDIVIDUAL QUEUES  *********************
*                                                                    *
*       SOURCE STATION :   SRC          ARRIVAL RATE :    0.556      *
*                                                                    *
*          L: LOWER BOUND;           A: APPROXIMATION                *
*                                                                    *
**********************************************************************
*   NAME   *   LOAD    * LENGTH(L) *RESPONSE(L)* LENGTH(A) *RESPONSE(A)*
**********************************************************************
* FORK     *  0.500    *    0.628  *    1.130  *    1.000  *    1.800  *
* PROC2    *  0.006    *    0.006  *    0.010  *    0.006  *    0.010  *
* PROC1    *  0.778    *    1.903  *    3.425  *    3.500  *    6.300  *
* JOINUP   *  0.556    *    0.759  *    1.365  *    1.250  *    2.250  *
* FINISH   *  0.667    *    1.144  *    2.059  *    2.000  *    3.600  *
**********************************************************************



LOWER BOUND OF THE RESPONSE TIME:

DISTRIBUTION FUNCTION :
RINF(x)=    1.0000 x^{ 0} e^{    0.0000 x }
 +                9.9131 x^{ 0} e^{   -0.4857 x }
 -              7.5549 x^{ 0} e^{   -0.7324 x }
 -              6.2224 x^{ 0} e^{   -0.2920 x }
 +              2.8642 x^{ 0} e^{   -0.8853 x }

FIRST MOMENT =              7.9785
SECOND MOMENT =             82.7641
VARIANCE =             19.1075


******** HEURISTIC RESULTS ON THE RESPONSE TIME ********

DISTRIBUTION FUNCTION :
RHEUR(x)=    1.0000 x^{ 0} e^{    0.0000 x }
 +              7.1111 x^{ 0} e^{   -0.2778 x }
 -              4.6296 x^{ 0} e^{   -0.4444 x }
 -              5.0815 x^{ 0} e^{   -0.1587 x }
 +              1.6000 x^{ 0} e^{   -0.5556 x }

FIRST MOMENT =             13.9500
SECOND MOMENT =            255.5552
VARIANCE =             60.9523

              MEMORY USED:       6734 WORDS OF 4 BYTES
               (  0.67  % OF TOTAL MEMORY)
     33
     34
     35
     36 & 2) Dual stage split-match network
     37
     38 /DECLARE/ QUEUE PROC3, PROC4, MID_JOIN;
     39
     40 /STATION/ NAME = PROC1;
     41           SPLIT = (PROC3, C3, 1, PROC4, C4, 1);
     42
     43 /STATION/ NAME = PROC3;
     44           SERVICE = EXP (0.5);
     45           TRANSIT = MID_JOIN;
     46
     47 /STATION/ NAME = PROC4;
     48           SERVICE = EXP (0.9);
     49           TRANSIT = MID_JOIN;
     50
     51 /STATION/ NAME = MID_JOIN;
     52           MATCH = (PROC1):(C3, 1, C4, 1) C1;
     53           SERVICE = EXP (1.0);
     54           TRANSIT = JOINUP;
     55
     56 /EXEC/ SOLVE ("SPLITMAT");
 - BOUNDS FOR SPLIT/MATCH SYSTEMS ("SPLITMAT") -
```

```
******************  RESULTS ON INDIVIDUAL QUEUES  **********************
*                                                                      *
*       SOURCE STATION :   SRC          ARRIVAL RATE :    0.556        *
*                                                                      *
*          L: LOWER BOUND;           A: APPROXIMATION                  *
*                                                                      *
************************************************************************
*   NAME   *   LOAD    * LENGTH(L) *RESPONSE(L)* LENGTH(A) *RESPONSE(A)*
************************************************************************
* FORK     *  0.500    *    0.628  *    1.130  *    1.000  *    1.800  *
* PROC2    *  0.006    *    0.006  *    0.010  *    0.006  *    0.010  *
* PROC1    *  0.778    *    1.903  *    3.425  *    3.500  *    6.300  *
* PROC4    *  0.500    *    0.628  *    1.130  *    1.000  *    1.800  *
* PROC3    *  0.278    *    0.287  *    0.516  *    0.385  *    0.692  *
* MID_JOIN *  0.556    *    0.759  *    1.365  *    1.250  *    2.250  *
* JOINUP   *  0.556    *    0.759  *    1.365  *    1.250  *    2.250  *
* FINISH   *  0.667    *    1.144  *    2.059  *    2.000  *    3.600  *
************************************************************************


LOWER BOUND OF THE RESPONSE TIME:

DISTRIBUTION FUNCTION :
RINF(x)=    1.0000 x^{ 0} e^{    0.0000 x }
 +              68.9044 x^{ 0} e^{   -0.4857 x }
 +             174.8680 x^{ 0} e^{   -0.7324 x }
 -              33.4597 x^{ 1} e^{   -0.7324 x }
 +               0.0183 x^{ 0} e^{   -1.9390 x }
 -             228.7079 x^{ 0} e^{   -0.8853 x }
 -               0.0013 x^{ 0} e^{   -2.8244 x }
 -              16.0815 x^{ 0} e^{   -0.2920 x }
 -              12.1464 x^{ 1} e^{   -0.8853 x }

FIRST MOMENT =              10.6349
SECOND MOMENT =             135.2399
VARIANCE =              22.1377


******** HEURISTIC RESULTS ON THE RESPONSE TIME ********

DISTRIBUTION FUNCTION :
RHEUR(x)=    1.0000 x^{ 0} e^{    0.0000 x }
 +              39.3823 x^{ 0} e^{   -0.2778 x }
 +              64.6735 x^{ 0} e^{   -0.4444 x }
 -              10.6147 x^{ 1} e^{   -0.4444 x }
 +               0.0036 x^{ 0} e^{   -1.4444 x }
 -              93.6983 x^{ 0} e^{   -0.5556 x }
 -               0.0004 x^{ 0} e^{   -2.0000 x }
 -              11.3608 x^{ 0} e^{   -0.1587 x }
 -                3.5556 x^{ 1} e^{   -0.5556 x }

FIRST MOMENT =             18.1924
SECOND MOMENT =            399.9451
VARIANCE =             68.9823

              MEMORY USED:       7712 WORDS OF 4 BYTES
               (  0.77  % OF TOTAL MEMORY)
     57
     58 /END/
```

5.2.13.5 Tracing facilities

No specific tracing facilities are provided with this solver. The TRACE option is ignored.

5.3 The Markovian Solver

5.3.1 Overview

The Markovian solver of QNAP2 may be used theorically for any model that may be mapped onto a first order markovian process with a finite number of states. The solver accepts the model description as input and performs the following operations:

- verifies that the solution method requirements are met and chooses a representation for the state of the model,
- computes all the possible states of the model, and their transition probabilities,
- computes the stationary probability of each of them,
- computes the performance criteria of the different model components (response time, ocupation rate, marginal probabilities. . .).

The solver computes exact values of the performance criteria characterizing the steady state of the model. Theoretically it may be used with a wide range of models (general service distributions, various scheduling disciplines, synchronization,. . .). In practice, its usage is limited by the number of states generated during the analysis of a model and therefore by its requirements in memory space and computation power.

The Markovian solver is activated by the procedure MARKOV.

5.3.2 Bibliography

STEWART, W. J. *MARCA: Markov Chain Analyser.* Rapport IRISA, No 45, Juin 1976, 127 pages.
STEWART, W. J. *A Comparison of Numerical Techniques in Markov Modeling.* C.ACM 21, 2, Feb. 1978, pp. 144-152.
SAAD, Y. *Krylov subspace Methods for Solving large Unsymmetric Linear Systems.* Mathematics of Computation 37, 155, July 1981, pp. 105-126.
SAAD, Y. *Variation on Arnoldi method for computing eigenelements of large unsymmetric matrices.* Linear agebra and applications 34, 1980, pp. 269-295.

5.3.3 Application conditions

5.3.3.1 Network topology

The network must be a closed network (no source nor exit queue). It may contain one or several customer classes.

5.3.3.2 Station types

The following station types are allowed:

- simple servers,
- multiple servers,
- infinite server,
- simple queue (station without service).

Resources or semaphores are not allowed.

5.3.3.3 Scheduling

The following scheduling disciplines are allowed:

- FIFO or LIFO with or without priorities, with or without preemption,
- PS (processor sharing).

Finite quantum is not allowed.

5.3.3.4 Services

Each service should include only one single work demand which may be distributed according to a general distribution. The work demand procedure may be followed or preceded by statements defining some dependencies (see below).
The permitted distributions are:

EXP (m): exponential (m = mean),

HEXP (m,c2): hyper-exponential (m= mean, c2= squared coefficient of variation),

ERLANG(m,k): Erlang (m = mean, and k=steps (c2=1/k)),

CST (m): constant m, approximated by an ERLANG(m,5),

COX (t): Coxian law with coefficients in table t.

The characteristics (mean, variance,...) of the work demand may depend on the instantaneous number of customers in each station (globally or for each class). In this case the characteristics of the work demand should be computed in the service description just before the call to the work demand procedure (the instantaneous number of customers may be accessed by the CUSTNB function).

Example:

```
/DECLARE/ QUEUE A;

/STATION/ NAME    = A;
          SERVICE = IF CUSTNB (A) < 3
                       THEN EXP (2)
                       ELSE ERLANG (2,3);
```

If the number of customers in A is less than 3, the service is distributed according to an exponential distribution, otherwise it is distributed according to an Erlang distribution.

Example:

```
/DECLARE/ QUEUE A, B;
          CLASS X, Y;
          REAL S;

/STATION/ NAME    = A;
          SERVICE = BEGIN
                      S:= 0.43 + 3.45*CUSTNB(B,X);
                      EXP (S);
                    END;
```

The service demand of class A customers linearly depends on the number of class X customers in B.

5.3.3.5 Transitions

The general probabilistic transition mechanism applies in the normal way. For dynamic routing, the procedures TRANSIT or MOVE may be used. They must appear after the work demand procedure in the description of the service.

Example:

```
/DECLARE/ QUEUE A, B, C;
          REF QUEUE Q;

/STATION/ NAME    = A;
          SERVICE = BEGIN
                      EXP(2);
                      IF CUSTNB(B)=0
                         THEN Q:=B
                         ELSE Q:=C;
                      TRANSIT(Q);
                    END;
```

At service completion time the customers of A are forwarded to B if this station is empty, otherwise to C.

Example:

```
/DECLARE/ QUEUE A, B ,C;
          CLASS X, Y;

/STATION/ NAME       = A;
          TRANSIT(X) = B,1, C, 1;
          SERVICE(X) = BEGIN
                         CST (1.5);
                         IF CUSTNB(B,X)+CUSTNB(C,X) = 3
                            THEN TRANSIT(A);
                       END;
```

If both B and C stations contain 3 customers, the customer completing its service in A is sent again to A otherwise it is forwarded to B or C with equal probabilities.

5.3.3.6 Service rate

The service rate of every server of a station may depend on the instantaneous number of customers present in this station. It is expressed with the general form of the RATE parameter.

Example:

```
/DECLARE/ QUEUE A;

/STATION/ NAME    = A;
          RATE    = 1, 1.8, 2.4;
          SERVICE = EXP (5);
```

```
& is equivalent to:

/STATION/ NAME    = A;
          SERVICE = IF CUSTNB (A)=1
                       THEN EXP (5)
                       ELSE IF CUSTNB (A) = 2
                               THEN EXP (5/1.8)
                               ELSE EXP (5/2.4);
```

5.3.3.7 Instantaneous transitions

After a transition performed at service completion time it is possible to initiate other instantaneous transitions under conditions dependent on the current number of customers in each station (CUSTNB function).

These transitions should be described in a test sequence associated with the TEST parameter. Each possible transition may be described only by means of the MOVE procedure. The origin station should be a simple queue (i.e. a station without server). If the origin queue is empty the MOVE procedure is ineffective.

Example:

Finite capacity queue

```
/DECLARE/ QUEUE A, B, W;

/STATION/ NAME    = A;
          TRANSIT = W;
          SERVICE = EXP (2);

/STATION/ NAME    = B;
          TRANSIT = A;
          SERVICE = HEXP(4,16);

/CONTROL/ TEST    = IF CUSTNB (B)<3 THEN MOVE (W,B);
```

As long as station B contains more than 3 customers the customers coming from A wait in the queue W. If a free place exists in B a customer of W is forwarded to B.

Example:

Combined services

```
/DECLARE/ QUEUE A, B, B1, B2;

/STATION/ NAME    = A;
          TRANSIT = B;
          SERVICE = EXP (1);

/STATION/ NAME    = B1;
          TRANSIT = B2;
          SERVICE = ERLANG (2.5, 2);

/STATION/ NAME    = B2;
```

```
                TRANSIT = A;
                SERVICE = HEXP (1, 4);

     /CONTROL/ TEST = IF CUSTNB (B1) + CUSTNB (B2) = 0
                          THEN MOVE (B, B1);
```

As soon as both B1 and B2 stations are empty a customer is sent from station B to B1. Therefore the B1 and B2 stations contain always only one customer at a time. This station is therefore equivalent to a station in which the service would be made of an Erlang demand followed by a hyper-exponential demand.

5.3.4 Results

The Markovian solver method computes all the performance criteria chararacterizing the steady state of the model together with the marginal probabilities of each station, globally or for each customer class.

5.3.5 Control of numerical convergence

The convergence of the numerical algorithms used to compute the steady-state probabilities may be controlled by the four parameters of the CONVERGENCE parameter of the CONTROL command:

parameter 1: maximum number of iterations (default value 100),

parameter 2: precision of the termination test (default value 1E-6),

parameter 3: number of test vectors used (default value 10),

parameter 4: ratio between the number of non-null entries in the state transition matrix and the number of states (default value 5).

5.3.6 Tracing facilities

The TRACE option (OPTION parameter of the CONTROL command) produces the following intermediate results:

- setting of the parameters used for the control of the numerical convergence,
- list of the states of the model and of the transition rates from one state to another,
- total number of states and number of non-zero elements in the transition matrix, convergence results at each iteration,
- stationary probabilities of the states.

Example:

```
     /DECLARE/ QUEUE A,B;
               INTEGER N;
               CLASS X,Y;

     /STATION/ NAME     = A;
               SCHED    = PRIOR,PREEMPT;
               INIT     = N;
```

```
                    PRIOR(X) = 1;
                    PRIOR(Y) = 2;
                    TRANSIT  = B;
                    SERVICE  = EXP(1.);

          /STATION/ NAME    = B;
                    SERVICE = EXP(3.);
                    TRANSIT = A;

          /CONTROL/ OPTION  = TRACE;

          /EXEC/    BEGIN
                      N:=1;
                      MARKOV;
                    END;
```

intermediate results:

```
 ITMAX= 100 EPS= 0.100E-05 M= 10
 ALPHA= 0.500E+01 NMAX= 1200 NAMAX= 6000

 STATES OF THE SYSTEM

 ... STATE NUMBER    1
   STATION A        CUSTNB= 2 STAGE: 0
           POSITION: 1 CLASS:Y        STAGE: 0
           POSITION: 2 CLASS:X        STAGE: 0
   STATION B        CUSTNB= 0 STAGE: 0


 ... STATE NUMBER    5
   STATION A        CUSTNB= 0 STAGE: 0
   STATION B        CUSTNB= 2 STAGE: 0
           POSITION: 1 CLASS:X        STAGE: 0
           POSITION: 2 CLASS:Y        STAGE: 0
                                     RATE FROM 4 TO 5=0.1000E+01
                                     RATE FROM 4 TO 1=0.3333E+00
                                     RATE FROM 5 TO 2=0.3333E+00

 TOTAL NUMBER OF STATES =         5
 NUMBER OF NON-ZERO ELEMENTS IN MATRIX =        12

 * * * * * * * * * *ITER= 1 * * * * * * * * * *
  ((CTR. RQIO ITS= 7V.P..
  & RES.    0.100000084983449D+01    0.197122871716286D-11))
  ITER= 1
  RESIDU=    0.36304150152540D-05
  V.P. =     0.100000084983449D+01


   2 ITERATIONS
```

```
*** PROBABILITY OF THE STATES ***
1 TO    5...
0.7692D-01 0.1319D+00 0.3956D+00 0.9890D-01 0.2967D+00
```

(the standard report is omitted).

5.4 Simulation

5.4.1 Overview

The SIMUL procedure of QNAP2 performs a discrete event simulation of the model under study. The simulation reproduces on the time scale of the model the dynamic behaviour of the different model components (stations, customers, variables,...). This behaviour is considered only at the instants at which an event occurs in the model (i.e. customer transition, service beginning,...).

Random number generators are used to reproduce the stochastic behaviour of the model components (service distribution times, probabilistic transitions,...).

The behaviour of the simulated model is automatically monitored during the simulation run in order to produce statistical estimations of performance criteria. Two types of statistical results can be computed:

Queue statistics are computed on performance criteria directly related to queues: service time, response time, blocked time, utilization rate, number of customers, throughput.

User statistics are computed on user-defined variables, called *watched* variables. The simulator tracks the watched variables whenever they are modified by the algorithmic code.

The simulator can be used to analyze steady-state as well as transient-state behaviour:

Global statistics are computed over the entire simulation run, except time intervals where the statistics are disabled in order to eliminate unwanted samples.

These statistics are computed under the assumptions that the model would reach a steady-state if the simulation were run during an infinite period of time. Therefore, the interpretation of the results of a simulation run should be carried out with respect to this basic assumption.

The longer the simulation run, the better the statistical estimations. In order to check the accuracy of the results of a simulation run, QNAP2 provides facilities for computing and editing the confidence intervals of the statistical estimations.

Partial statistics are computed over specific time periods.

These statistics are used to analyze transient states: startup period, transitions between two steady states, or systems with a non-stationary behaviour (e.g., daily activity profile).

Discrete event simulation may be used for the analysis of any model specified in the language of QNAP2 (with a few exceptions described in the next section). Therefore, the simulation and language features of QNAP2 provide facilities comparable to those found in general purpose simulation languages (GPSS, SIMULA, SIMSCRIPT,...).

5.4.2 Bibliography

LEROUDIER, J. and PARENT, M. *Discrete Event Simulation of Computer System for Performance Evaluation.* Rapport de recherche IRIA 177, Juin 1976.

5.4.3 Application stipulations

All the mechanisms described in chapter 4 may be used including processor sharing scheduling (PS scheduling). The principle is that all the customers in the station with the `PS` scheduling share the servers.
Restriction: Synchronization procedures cannot be used in this case (`P`, `WAIT`, `WAITAND`, `WAITOR`, `JOIN`, `JOINC`). If synchronizations are mandatory, PS can be approximated by a QUANTUM scheduling with a small quantum value, compared to the mean service time.

5.4.4 Control of simulation length

The TMAX parameter specifies the maximum duration of a simulation run (measured on the time scale of the model). When the current time reaches the value TMAX the simulation is stopped.

However, the simulation may be stopped before this limit by means of a two different mechanims:

- When the STOP procedure is explicitely called
- When a requested accuracy has been obtained

5.4.4.1 Explicit stop

The STOP procedure can be called in any algorithmic code sequence during simulation. By this means one can stop a simulation run as soon as a given condition is met in the model.

Example:

```
/DECLARE/ QUEUE A;

/STATION/ NAME    = A;
          SERVICE = BEGIN
                      EXP (23.5);
                      IF A.NBOUT >500 THEN STOP;
                    END;

/CONTROL/ TMAX = 1.E4;
```

In this case the simulation stops either after 10000 time units or as soon as A has performed more than 500 services (The NBOUT attribute counts the number of customers having left the station since the start of simulation).

The STOP procedure ends the simulation in a "clean" way: the statistical results are computed and displayed normally. The statements following the SIMUL procedure are executed normally. In order to stop a model and exit from QNAP2, use the ABORT procedure.

5.4.4.2 Accuracy control

QNAP2 can compute *confidence intervals* on mean values. A confidence interval is an indication on the quality of the estimation of the mean from a finite set of samples. A small confidence interval indicates that the simulation was long enough to reach a steady state and provide reliable statistics. Confidence intervals can be computed on queue statistics as well as user statististics.

In order to spare computer time, the user can request to stop the simulation as soon as a given accuracy has been reached.

Example:

Accuracy control is requested on all statistics of queue SYSTEM (service time, response time, number of customers, blocked time). The simulation stops when the confidence interval falls below 10% of the mean estimate.

```
 SIMULOG   ***  QNAP2  ***  ( 01-04-95  ) V 9.2
 (C)  COPYRIGHT BY CII HONEYWELL BULL AND INRIA, 1986
```

```
    /DECLARE/ QUEUE USER, SYSTEM;

    /STATION/ NAME = USER;
              INIT = 10;
              SERVICE = EXP (1);
              TRANSIT = SYSTEM;

    /STATION/ NAME = SYSTEM;
              SERVICE = ERLANG (1, 5);
              TRANSIT = USER;

    $MACRO SYSACC (RESULT)
       GETSTAT:RESULT:MEAN (SYSTEM) : 8,
       " +/-  ", 100 * GETSTAT:RESULT:ACCURACY (SYSTEM) /
       GETSTAT:RESULT:MEAN (SYSTEM) : 8, " % "
    $END

    /CONTROL/ TMAX = 10000;
              OPTION = NRESULT;
              PERIOD = 500;
              ESTIMATION = REGENERATION;
              TEST = BEGIN
                       SAMPLE;
                        PRINT (TIME, $SYSACC (CUSTNB), $SYSACC (RESPONSE));
                     END;

    /EXEC/ BEGIN
              SETSTAT:QUEUE (SYSTEM);
              SETSTAT:ACCURACY (SYSTEM);
              SETSTAT:PRECISION (SYSTEM, 0.10);
           END;

    /EXEC/ SIMUL;
500.0     4.15136 +/-  -24.0885 %  4.40630 +/-  -22.6948 %
1000.     4.04608 +/-   5.10046 %  4.39679 +/-   0.43161 %
1500.     4.20035 +/-   7.73749 %  4.55322 +/-   6.74523 %
2000.     4.29807 +/-   6.96052 %  4.65108 +/-   6.20286 %
2500.     4.46929 +/-   9.12489 %  4.82378 +/-   8.38488 %
3000.     4.56533 +/-   8.37858 %  4.94440 +/-   8.20136 %
3500.     4.84544 +/-   13.1489 %  5.19920 +/-   11.5783 %
4000.     4.77794 +/-   11.8754 %  5.16599 +/-   10.2174 %
4500.     4.90095 +/-   11.3336 %  5.26626 +/-   9.53640 %
5000.     4.90864 +/-   10.1259 %  5.30314 +/-   8.60059 %
5500.     4.85675 +/-   9.49091 %  5.25512 +/-   8.07207 %
6000.     4.97648 +/-   9.68147 %  5.35181 +/-   7.99374 %
6500.     4.90010 +/-   9.54655 %  5.28072 +/-   7.93696 %
    /END/
```

This example illustrates some of the features of the precision control facility.

- The accuracy of a result is considered satisfactory when:
  1. The confidence interval is lower than the requested threshold for the last two samples.
  2. The confidence interval is decreasing.
- The simulation is stopped when *all* results are satisfactory.

5.4.5 TEST sequence

A test sequence executed periodically during a simulation run may be specified with the TEST parameter. This sequence may include statements testing the model state and performing various actions. In the test sequence one may:

- access model variables and perform various printing and tests,
- print the accumulated statistics about the model behaviour (OUTPUT procedure),
- stop a simulation run (STOP procedure),
- define a regeneration point (call to the SAMPLE or SETSTAT:SAMPLE procedures) when the regeneration method is used.

However, a test sequence may not include operations on customers (e.g. creation or transition) nor synchronizations.

If the TEST parameter is not specified, no action is performed (i.e. default value: `TEST = ;`). The test sequence specified with the TEST parameter is executed periodically according to the interval value specified in the PERIOD parameter of the /CONTROL/ command.
There is no default value for the parameter PERIOD. If the parameter PERIOD is not defined the test sequence is not activated.

- If PERIOD=0, the test sequence is executed at each occurence of an event in the model. This may be used to systematically test the model state. Note that this facility is costly and should only be used when necessary.
- PERIOD = ; cancels any previous definition of the parameter PERIOD.
- TEST = ; cancels any previous definition of the TEST parameter.

Example 1:

```
/CONTROL/ TMAX   = 1.E6;
          PERIOD = 1.E5;
          TEST   = OUTPUT;
```

After each interval of 100 000 time units the OUTPUT procedure is called (printing of the cumulated statistics).
Example 2:

```
/DECLARE/ QUEUE CPU, DISK;
          INTEGER NSAMPLE;

/CONTROL/ TMAX   = 1.E6;
          PERIOD = 0;
          ESTIM  = REGENERATION;
          TEST   = IF (CPU.NB + DISK.NB = 0)
                      THEN BEGIN
                             SAMPLE;
                             NSAMPLE:= NSAMPLE + 1;
                             IF NSAMPLE = 100 THEN STOP;
                           END;
```

The state of the CPU and DISK stations is continuously tested (i.e. whenever something happens in the model). If the stations are both empty, a regeneration point is created and the simulation is stopped when 100 regeneration points have occured.

5.4.6 Estimation of confidence intervals

Simulating a stochastic model consists in generating one or several finite trajectories of the model using random number streams. The required characteristics of the equilibrium (or transient) behaviour of the model are then estimated from the information contained in the simulated trajectories. Because of the finite number and length of the generated trajectories the estimates can provide only approximate values of the required characteristics. Therefore, some indications on the accuracy of the estimates are needed.

The standard method of estimating the accuracy of a statistical estimate is to compute confidence intervals. We say that we have a confidence interval for the estimate of a characteristic if we can state that the exact value of the estimated characteristic is within the confidence interval with a specified probability. This probability, expressed in percent, is called the confidence level. In QNAP2, confidence intervals are produced with a confidence level of 95 %. The confidence intervals produced are themselves estimates because the computation of exact confidence intervals would require an a-priori knowledge of the estimated characteristics.
QNAP2 includes three methods for confidence interval estimation:

1. the replication method,
2. the regeneration method,
3. the spectral method.

These methods rely on various assumptions described in the following three sub-sections and have different characteristics in term of computational complexity and memory requirement.

The selection of the confidence interval method and of the parameters specific to each method is made within the /CONTROL/ command with the following parameters:

ESTIMATION: defines the confidence interval method used (REPLICATION, REGENERATION, SPECTRAL),

ACCURACY: specifies the list of queues for which confidence intervals are to be produced,

TEST: introduces a test sequence, i.e. a process which is activated periodically during the simulation,

PERIOD: defines the period of activation of the test sequence,

CORRELATION: defines a list of queues for which autocorrelation functions are to be computed (applicable only to the regeneration method).

Alternately, the following procedures are available in the algorithmic language:

SETSTAT:ACCURACY specifies the statistical variables for which confidence must be computed

SETSTAT:CORRELATION specifies the statistical variables for which autocorrelation fucntions must be computed

The confidence intervals produced can be retrieved by specific result access functions during a simulation run or after completion of the simulation run. For example: CCUSTNB and GETSTAT:CUSTNB:ACCURACY both return the confidence interval on the mean number of customers in a queue. A complete list of these functions is available in the QNAP2 Reference Manual.

5.4.6.1 The replication method

Independent replications is the most straightforward method for obtaining confidence intervals. It consists in generating several sample path of the model studied, so that these sample path are statistically independent and identical. This is achieved by reseting the original initial state of the model at the beginning of each replication, and by using a different random number stream for each replication. In practice a single random number stream is used throughout the whole simulation run so that the random stream for the second replication begins where the stream for the first replication ended, and so on. This guarantees independent random number streams.

The value of the parameter ESTIMATION for the regeneration method is REPLICATION. The number of replications is given by the argument of the keyword REPLICATION. This argument is mandatory.

Example:

```
/DECLARE/ QUEUE SOURCE,SYST;

/STATION/ NAME    = SOURCE;
          TYPE    = SOURCE;
          TRANSIT = SYST;
          SERVICE = EXP (1);

/STATION/ NAME    = SYST;
          TRANSIT = OUT;
          SERVICE = HEXP (0.7, 5);

/CONTROL/ TMAX     = 1500;
          ACCURACY = SYST;
          ESTIM    = REPLICATION (10);

/EXEC/    SIMUL;
```

```
     *** SIMULATION WITH REPLICATION METHOD ***
     ... MEAN SIMULATION TIME =    1500.000
         NUMBER OF REPLICATIONS =    10
         CONFIDENCE LEVEL = 0.95
*******************************************************************
* NAME     * SERVICE   * BUSY PCT * CUST NB  * RESPONSE * SERV NB  *
*******************************************************************
*          *           *          *          *          *          *
* SOURCE   * 1.009     * 1.000    * 1.000    * 1.009    * 1486.    *
*          *           *          *          *          *          *
* SYST     * .7277     * .7206    * 6.507    * 6.579    * 1482.    *
* +/-      *0.2151E-01*0.2821E-01* 1.083    * 1.024    *          *
*          *           *          *          *          *          *
*******************************************************************
     ... END OF SIMULATION ...


                  MEMORY USED:        4555 WORDS OF 4 BYTES
                   ( 1.75 % OF TOTAL MEMORY)
```

Note:

In order to limit the cost of the simulation run a trade-off has to be found between the length of each replication and the number of replications. If the steady-state behaviour of the model is looked for, each replication should be long enough to ensure that the steady-state is effectively reached and then it is better to have longer replications and a smaller number of replications. The reverse is true if the simulation is run to analyse the transient behaviour of the model. Then shorter replications and a larger number of replications are preferable.

5.4.6.2 The regeneration method

The basic principle is to split the simulation run into several successive intervals, so that the model behaviour in these intervals be statistically equivalent and independent. Thus there are as many independent sub-simulations as there are intervals and the results within each simulation sub-run are considered as samples of independent identically distributed random variables; these variables are used to estimate the confidence intervals and the final results.

In practice, the independence hypothesis may be satisfied if the model corresponds to a stochastic regenerative process. In this case the model contains one or more special states, called regeneration states. Whenever the system returns to one of these states a regeneration point is obtained, i.e. a point where the past behaviour of the system has no more influence on its future behaviour. The regeneration points may be used to split the simulation duration into intervals which satisfy the statistical independence hypothesis.

The determination of exact regeneration intervals is not, in general, feasible on reasonably complex models. The simplest case is the case of Markovian models in which every state is a regeneration state. Similarly, in an open network model having a Poisson arrival process the state “all the stations empty” is generally a regeneration state.

The user may define regeneration points by testing a condition defining the regeneration states and by explicitly requesting the computation of partial statistics on the last interval if the test is satisfied. The more general way to test a condition is to program the test in a special process, called a test sequence, which is activated periodically with a period specified by the user with the parameter PERIOD. If a zero period is explicitly specified the test sequence is executed whenever an event is processed by the simulator: the condition is then "continuously" tested. A test sequence is defined within the /CONTROL/ command by the TEST parameter.

Whenever the specified regeneration state is reached, the computation of the partial statistics on the last interval must be requested explicitly using the QNAP2 SAMPLE procedure. The function of SAMPLE is the computation of partial statistics on the last regeneration interval and of global statistics and confidence intervals using the information collected from the beginning of the simulation up to the current regeneration point. In this way partial results on the simulation are available before the simulation run is completed. However, these intermediate results are not output unless explicitly requested by a call to the procedure OUTPUT or through the results access functions.

The regeneration method is specified by the parameter ESTIMATION = REGENERATION.

Example:

```
             /DECLARE/ QUEUE SOURCE,SYST;

             /STATION/ NAME    = SOURCE;
                       TYPE    = SOURCE;
                       SERVICE = EXP (1);
                       TRANSIT = SYST;

             /STATION/ NAME    = SYST;
                       TRANSIT = OUT;
                       SERVICE = HEXP (0.7, 5);

             /CONTROL/ TMAX     = 15000;
                       ACCURACY = SYST;
                       ESTIM    = REGENERATION;
                       PERIOD   = 0.;
                       TEST     = IF SYST.NB = 0 THEN SAMPLE;

             /EXEC/    SIMUL;

***SIMULATION WITH REGENERATIVE METHOD ***

... TIME =    14998.52 , NB SAMPLES = 4363 , CONF. LEVEL = 0.95
********************************************************************
* NAME     * SERVICE  * BUSY PCT * CUST NB  * RESPONSE * SERV NB  *
********************************************************************
*          *          *          *          *          *          *
* SOURCE   * 1.007    * 1.000    * 1.000    * 1.007    *     14895*
*          *          *          *          *          *          *
* SYST     * .7131    * .7082    * 5.676    * 5.716    *     14894*
* +/-      *0.2458E-01*0.2680E-01* 1.323    * 1.302    *          *
*          *          *          *          *          *          *
********************************************************************
... END OF SIMULATION ...


                  MEMORY USED:       4647 WORDS OF 4 BYTES
                   ( 1.78 % OF TOTAL MEMORY)
```

Another way to test a regeneration state is to program this test in a service, provided that the condition is met only when this service is active. This is in general a more efficient approach because the condition is tested less frequently than in the previous case.

Example:

```
              /DECLARE/ QUEUE SOURCE,SYST, MONITOR;

              /STATION/ NAME    = SOURCE;
                        TYPE    = SOURCE;
                        TRANSIT = SYST;
                        SERVICE = EXP (1);

              /STATION/ NAME    = SYST;
                        TRANSIT = MONITOR;
                        SERVICE = HEXP (0.7, 5);

              /STATION/ NAME    = MONITOR;
                        TRANSIT = OUT;
                        SERVICE = IF SYST.NB = 0 THEN SAMPLE;

              /CONTROL/ TMAX     = 15000;
                        ACCURACY = SYST;
                        ESTIM    = REGENERATION;

              /EXEC/    SIMUL;
```

```
***SIMULATION WITH REGENERATIVE METHOD ***

... TIME =    15000.00 , NB SAMPLES = 4364 , CONF. LEVEL = 0.95
*******************************************************************
* NAME   * SERVICE * BUSY PCT * CUST NB * RESPONSE * SERV NB *
*******************************************************************
*          *          *          *          *          *          *
* SOURCE   * 1.007    * 1.000    * 1.000    * 1.007    *     14896*
*          *          *          *          *          *          *
* SYST     * .7131    * .7081    * 5.676    * 5.715    *     14896*
* +/-      *0.2458E-01*0.2680E-01* 1.323    * 1.302    *          *
*          *          *          *          *          *          *
* MONITOR  *0.0000E+00*0.0000E+00*0.0000E+00*0.0000E+00*     14896*
*          *          *          *          *          *          *
*******************************************************************
... END OF SIMULATION ...


               MEMORY USED:       4728 WORDS OF 4 BYTES
                ( 1.81 % OF TOTAL MEMORY)
```

The results obtained (point estimates and confidence intervals) are identical to the previous ones. Indeed the two simulation runs use the same random number stream, have the same length and the same regeneration points are used. The difference between the two simulations lies in the way the regeneration points are detected (and of course, in the computation cost).

In many situations approximate regeneration points for a given model can be derived from the existence of exact regeneration states for a simpler version of the model studied. For example, in the previous model the existence of a regeneration state is related to the Poisson assumption on the arrival process. If the arrival process is not Poisson, the state "system empty" is no longer a regeneration point. However, it may still be considered as an approximate regeneration point as in the next example where hyper-exponential inter-arrival times are assumed.

However, one may question the validity of the confidence intervals which are obtained because of the approximate regeneration points used. Indeed, the statistical independence of the intervals must be thoroughly assessed before meaningful interpretations of the confidence intervals are made. For this purpose one can request the computation of the autocorrelation functions on the basic quantities used in the derivation of the standard performance criteria. This is done with the CORRELATION parameter of the /CONTROL/ command. The value of the parameter CORRELATION is the list of queues for which autocorrelation functions are requested. The maximum order of the autocorrelation functions is defined by an integer following the list of queues (the default value for the order is 5).

Example:

```
            /DECLARE/ QUEUE SOURCE,SYST, MONITOR;

            /STATION/ NAME    = SOURCE;
                      TYPE    = SOURCE;
                      TRANSIT = SYST;
                      SERVICE = HEXP (1,5);

            /STATION/ NAME    = SYST;
                      TRANSIT = MONITOR;
                      SERVICE = HEXP (0.7, 5);

            /STATION/ NAME    = MONITOR;
                      TRANSIT = OUT;
                      SERVICE = IF SYST.NB = 0 THEN SAMPLE;

            /CONTROL/ TMAX        = 15000;
                      ACCURACY    = SYST;
                      ESTIM       = REGENERATION;
                      CORRELATION = SYST;

            /EXEC/    SIMUL;
```

```
***SIMULATION WITH REGENERATIVE METHOD ***

... TIME =    15000.00 , NB SAMPLES = 2459 , CONF. LEVEL = 0.95
*******************************************************************
* NAME     * SERVICE  * BUSY PCT * CUST NB  * RESPONSE * SERV NB  *
*******************************************************************
*          *          *          *          *          *          *
* SOURCE   * 1.012    * 1.000    * 1.000    * 1.012    *     14823*
*          *          *          *          *          *          *
* SYST     * .7102    * .7019    * 8.790    * 8.896    *     14822*
* +/-      *0.2495E-01*0.3629E-01* 2.168    * 2.040    *          *
*          *          *          *          *          *          *
* MONITOR  *0.0000E+00*0.0000E+00*0.0000E+00*0.0000E+00*     14822*
*          *          *          *          *          *          *
*******************************************************************

             AUTOCORRELATION FUNCTIONS ON QUEUE MEASURES

*************************************************************************
*                       AUTOCORRELATION ON QUEUE SYST                   *
*************************************************************************
* ORDER  * BLOC  * SERV   * BUSY   * QUEUE  *  RES   * NB     * BLOCKED *
*        * SIZE  * TIME   * TIME   * LENGTH *  TIME  * SERVED * TIME    *
*************************************************************************
*    1   * 0.021 * -0.003 * -0.003 * -0.005 * -0.005 * -0.009 * 0.000   *
*    2   *-0.006 * -0.009 * -0.009 *  0.001 *  0.001 * -0.009 * 0.000   *
*    3   * 0.001 * -0.005 * -0.005 * -0.003 * -0.003 * -0.006 * 0.000   *
*    4   * 0.008 *  0.003 *  0.003 * -0.003 * -0.003 *  0.010 * 0.000   *
*    5   *-0.016 * -0.015 * -0.015 * -0.006 * -0.006 * -0.012 * 0.000   *
*************************************************************************


                END OF AUTOCORRELATION COMPUTATIONS

... END OF SIMULATION ...


              MEMORY USED:       5201 WORDS OF 4 BYTES
               ( 1.99 % OF TOTAL MEMORY)
```

The values of the autocorrelation coefficients confirm the validity of the state "system empty" as a regeneration point. However, it can be observed that the confidence interval obtained is significantly larger than with the previous model: because of the hyper-exponential inter-arrival process the variance of the response time is greater.

Without theoretical results determining the regeneration states of the model, one will choose artificial regeneration points. For example, spliting the simulation run into fixed length sub-runs produces intervals which satisfy the independence hypothesis if the length of the intervals is large enough. The end of each interval can then be considered as an approximate regeneration point.

The length of the intervals should be large enough so that within an interval the behaviour of the model be as independent as possible of its behaviour in the previous interval. On the other hand, the number of intervals should be large enough to ensure an accurate estimation of the confidence intervals. A useful rule of thumb is to have 100 fixed intervals.

This method can be simply implemented by causing the periodic activation of the procedure SAMPLE as shown in the following example:

Example:

```
                  /DECLARE/ QUEUE SOURCE,SYST;
                  /STATION/ NAME    = SOURCE;
                            TYPE    = SOURCE;
                            SERVICE = EXP (1);
                            TRANSIT = SYST;

                  /STATION/ NAME    = SYST;
                            SERVICE = HEXP (0.7, 5);
                            TRANSIT = OUT;

                  /CONTROL/ TMAX        = 15000;
                            ACCURACY    = SYST;
                            ESTIM       = REGENERATION;
                            PERIOD      = 150;
                            TEST        = SAMPLE;
                            CORRELATION = SYST;

                  /EXEC/    SIMUL;

***SIMULATION WITH REGENERATIVE METHOD ***

... TIME =    15000.00 , NB SAMPLES =    100 , CONF. LEVEL = 0.95
*******************************************************************
* NAME     * SERVICE  * BUSY PCT * CUST NB  * RESPONSE * SERV NB  *
*******************************************************************
*          *          *          *          *          *          *
* SOURCE   * 1.007    * 1.000    * 1.000    * 1.007    *     14896*
*          *          *          *          *          *          *
* SYST     * .7131    * .7081    * 5.676    * 5.715    *     14896*
* +/-      *0.2493E-01*0.2421E-01* 1.079    * 1.046    *          *
*          *          *          *          *          *          *
*******************************************************************

              AUTOCORRELATION FUNCTIONS ON QUEUE MEASURES

*************************************************************************
*                        AUTOCORRELATION ON QUEUE SYST                  *
*************************************************************************
* ORDER  * BLOC  *  SERV  *  BUSY  * QUEUE  *  RES   * NB     * BLOCKED *
*        * SIZE  *  TIME  *  TIME  * LENGTH *  TIME  * SERVED * TIME    *
*************************************************************************
*    1   * 0.000 *  0.131 *  0.174 *  0.265 *  0.289 * -0.279 * 0.000   *
*    2   * 0.000 * -0.068 * -0.083 * -0.020 * -0.006 *  0.122 * 0.000   *
*    3   * 0.000 * -0.004 * -0.021 * -0.013 * -0.026 *  0.095 * 0.000   *
*    4   * 0.000 * -0.002 * -0.014 * -0.054 * -0.063 * -0.078 * 0.000   *
*    5   * 0.000 * -0.139 * -0.123 * -0.060 * -0.025 *  0.068 * 0.000   *
*************************************************************************


             END OF AUTOCORRELATION COMPUTATIONS
```

```
... END OF SIMULATION ...

             MEMORY USED:       4641 WORDS OF 4 BYTES
              ( 1.78 % OF TOTAL MEMORY)
```

The results show that the independence assumption is satisfied with this model with intervals of length 100.;

Note:

It is important to recall that the independence assumption between the consecutive intervals created by the regeneration points is essential for the correct estimation of confidence intervals. Meaningless confidence intervals would be produced if the independence is not verified.

5.4.6.3 The spectral method

The basic principle of the two previous methods is to build a set of independent and identically distributed samples and to apply standard statistical techniques for estimating confidence intervals. The spectral method is somewhat diferent in the sense that it does not rely on the independence assumption and applies to correlated samples provided that the identical distribution property is satisfied. Indeed, the correlation is explicitly taken into account in the estimation of the confidence intervals. As a consequence, the confidence intervals produced by the spectral method are most often larger than those produced by the regenerative method because of the effect of correlation in the variance of the point estimates.

The main advantage of the spectral method over the two previous methods is that the user needs not bother with the choice of specific parameters (number of replications, regeneration state,..): the method applies to the analysis of the stationary behaviour of any simulated model.

The value of the parameter ESTIMATION for the spectral method is ESTIMATION = SPECTRAL. The parameter of the spectral method can be specified by the argument of the keyword SPECTRAL. This parameter defines the value of the first measurement interval of the spectral method. It is optional and the default value TMAX/512 is used if no argument is given.

Example:

```
/DECLARE/ QUEUE SOURCE,SYST;

/STATION/ NAME    = SOURCE;
          TYPE    = SOURCE;
          TRANSIT = SYST;
          SERVICE = EXP (1);

/STATION/ NAME    = SYST;
          TRANSIT = OUT;
          SERVICE = HEXP (0.7, 5);

/CONTROL/ TMAX     = 15000;
          ACCURACY = SYST;

/EXEC/    SIMUL;
```

```
***SIMULATION WITH SPECTRAL METHOD ***

... TIME =    15000.00 , NB SAMPLES =    512 , CONF. LEVEL = 0.95
******************************************************************
* NAME     * SERVICE  * BUSY PCT * CUST NB  * RESPONSE * SERV NB  *
******************************************************************
*          *          *          *          *          *          *
* SOURCE   * 1.007    * 1.000    * 1.000    * 1.007    *     14896*
*          *          *          *          *          *          *
* SYST     * .7131    * .7081    * 5.676    * 5.715    *     14896*
* +/-      *0.2668E-01*0.2784E-01* 1.659    * 1.752    *          *
*          *          *          *          *          *          *
******************************************************************
... END OF SIMULATION ...


              MEMORY USED:       5864 WORDS OF 4 BYTES
               ( 2.25 % OF TOTAL MEMORY)
```

The results show that the confidence intervals obtained with the spectral method are somewhat larger than with the regeneration approach. This was to be expected because no assumption on the independence of the samples is made and as a consequence covariance terms are included in the estimate of the variance.

5.4.7 Parallel Replications

5.4.7.1 Presentation.

When the confidence intervals are computed by the replication method, the replications can be started in parallel on several workstations. The various files used by QNAP2 when running must be visible (by `NFS`) of each machine used. The QNAP2 installation directory and the user's "home directory" must also be installed (by `NFS`) on each machine.
The principle of parallel replications is as follows: the main QNAP2 executable (master) divides the running of the replications between several slave executables on each of the workstations used. To get different replications after each replication, the slave executables possess different random germs. These are read in the file `ranpar.dat` which is in the QNAP2 delivery directory. Communication between the master and slave processes is done by the `PVM` (Parallel Virtual Machine) software version 3.2:

```
          PVM 3.2:  Parallel Virtual Machine System 3.2
              University of Tennessee, Knoxville TN.
          Oak Ridge National Laboratory, Oak Ridge TN.
                  Emory University, Atlanta GA.
     Authors:  A. L. Beguelin, J. J. Dongarra, G. A. Geist,
   W. C. Jiang, R. J. Manchek, B. K. Moore, and V. S. Sunderam
                  (C) 1992 All Rights Reserved
```

Warning : The version of the `PVM` software used by QNAP2 is slightly different from the standard version, so as to allow for the environment variables specific to `MODLINE`. Make sure you use the version provided with QNAP2 when using it. This paragraph only gives the minimum necessary information for the use of parallel replications. It is not a `PVM` reference manual.
You can get more information about `PVM` by:

- email : `netlib@ornl.gov`
- reading the "Newsgroup" `comp.parallel.pvm`
- ftp : `netlib2.cs.utk.edu` (pvm directory)
- http : `http://www.epm.ornl.gov/pvm/pvm_home.html`

5.4.7.2 Installation - Startup.

Using the parallelization of replications requires the environment variable PVM_ROOT to be placed in the initialisation file of your "login shell". For example, if your login shell is csh, your .cshrc must contain the following line, if MODLINE is installed:

```
setenv PVM_ROOT $MODLINEHOME/bin/$ARCH/qnap2_tape
```

the next two lines, if MODLINE is not installed and if QNAP2 is installed (alone) in the /usr/local/qnap2 directory (for example):

```
setenv PVM_QNAP2 /usr/local/qnap2
setenv PVM_ROOT $PVM_QNAP2
```

The parallelization of replications is set off by the following command in the model QNAP2:

```
/CONTROL/
OPTION=SIMPAR;
```

and cancelled by the command (default option):

```
/CONTROL/
OPTION=NSIMPAR;
```

To actually get replications which start in parallel, the confidence intervals must obviously be computed by the replications method (see previous paragraph on the estimation of the confidence intervals):

```
/CONTROL/
ESTIMATION=REPLICATION(number);
```

5.4.7.3 Use - Warnings.

To start the replications (on a model containing `OPTION=SIMPAR;`), you must first start the communication interface with `PVM`. To do this, you start the `pvm` command in the QNAP2 installation directory, in the sub-directory `lib` (`$PVM_ROOT/lib/pvm`). A prompt then appears:

```
pvm>
```

You can then enter the `PVM` configuration commands. A full list is given at the end of the chapter. The useful commands are:

- `add station_name` - allows you to work (to start a process) on the station in question.
- `conf` - gives the list of stations that can be used (on which processes can be started).
- `delete station_name` - removes the station from the list of usable machines.
- `halt` - stops `PVM` and quits the interface.
- `kill number` - kills the process with the number specified.
- `kill -c number` - kills the child process of the process whose number has been specified.
- `ps -a` - gives the list of all the processes, with their associated numbers (`TID` column) and the number of their parent process (`PTID` column). The numbers are hexadecimal.
- `quit` - leaves the interface without killing the process.
- `reset` - kills all the processes without leaving the interface. Useful if, following an entry error, QNAP2 "slaves" continue to function.
- `help` - online help.

5.4.7.3.1 Example. Let us say you have a model `model.qnp` containing the replication parallelization commands, i.e.:

```
/CONTROL/
ESTIMATION=REPLICATION(number);
OPTION=SIMPAR;
```

and you want to run this model on the stations `joe`, `william` and `jack`. Let us say, also, that you are working on `joe`. Comments on the user operations are between the signs < and >.

```
john@joe% pvm              <start the interface on the machine
                            joe, which means that it is automatically
                            in the list of usable machines>
pvm> add william           <adds william to the list of stations>
1 successful
                    HOST     DTID
                 william    c0000
pvm> add jack
1 successful
                    HOST     DTID
                    jack    c0000
pvm> add joe               <error : you are trying to add
                            to the list of usable stations one which
                            is already there, in this case the one on
                            which you are working>
0 successful
                    HOST     DTID
```

```
                     joe Duplicate host
pvm> add averell           <error : you are using a station
                            you did not intend to use >
1 successful
                    HOST     DTID
                 averell    c0000
pvm> delete averell        <so remove this station from the list>
1 successful
                    HOST  STATUS
                 averell  deleted
pvm> conf
3 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
                     joe    40000     SUN4    1000
                 william    c0000     SUN4    1000
                    jack   100000     SUN4    1000
pvm> quit

pvmd still running.
john@joe%
```

Then you start the QNAP2 executable on the model in question (`model.qnp`). If, while the replications are running, you return to the PVM communication interface, you can see the list of the "slave" executables of QNAP2 which are running:

```
john@joe% pvm
pvm> ps -a
                    HOST        A.OUT      TID     PTID FLAG
                     joe            -    40003        - 0x04/c
                     joe      QNSLAV9    40004    40003 0x04/c
                 william      QNSLAV9    c0001    40003 0x04/c
                    jack      QNSLAV9   100001    40003 0x04/c
pvm>
```

The QNAP2 "master" has the number 40003 and the slaves the numbers 40004 c0001 and 100001. You can then stop PVM once the replications have been done:

```
pvm> halt
libpvm [t40005]: mxfer() EOF on pvmd sock
john@joe%
```

You can also exit by `quit`. In this case, PVM will continue to run as a background task.
Here is a simple example of the resulting file. You will observe that the 3 stations used (`joe`, `william` and `jack`) have not simulated the same number of replications (because `joe` was faster than `william`, itself faster than `jack`):

```
  SIMULOG   ***  QNAP2  ***  ( 01-04-95  ) V 9.2
  (C)  COPYRIGHT BY CII HONEYWELL BULL AND INRIA, 1986


      1 /DECLARE/ QUEUE SOURCE,SYST;
      2
      3 /STATION/ NAME    = SOURCE;
```

```
     4           TYPE    = SOURCE;
     5           TRANSIT = SYST;
     6           SERVICE = EXP (1);
     7
     8 /STATION/ NAME    = SYST;
     9           TRANSIT = OUT;
    10           SERVICE = HEXP (0.7, 5);
    11
    12 /CONTROL/ TMAX       = 150;
    13           ACCURACY   = SYST;
    14           ESTIMATION = REPLICATION (20);
    15           OPTION     = SIMPAR;
    16
    17 /EXEC/    SIMUL;

*** SIMULATION WITH REPLICATION METHOD ***
... MEAN SIMULATION TIME =         150.000
    NUMBER OF REPLICATIONS =    20
    CONFIDENCE LEVEL = 0.95
*******************************************************************
*   NAME   *  SERVICE * BUSY PCT *  CUST NB * RESPONSE *  SERV NB *
*******************************************************************
*          *          *          *          *          *          *
* SOURCE   * 1.004    * 1.000    * 1.000    * 1.004    * 148.1    *
*          *          *          *          *          *          *
* SYST     *0.6894    *0.6651    * 4.344    * 4.149    * 143.0    *
*   +/-    *0.5583E-01*0.5567E-01* 1.158    *0.9705    *          *
*          *          *          *          *          *          *
*******************************************************************
  3 slaves used for   20 replications
***************************************************************
joe                        :  11 replication(s),    55.0000%
william                    :  7 replication(s),     35.0000%
jack                       :  2 replication(s),     10.0000%
***************************************************************
... END OF SIMULATION ...


             MEMORY USED:       5922 WORDS OF 4 BYTES
              (  0.59  % OF TOTAL MEMORY)
    18
    19 /END/
```

5.4.7.3.2 Precautions for use. Because of the way it works, parallelization of replications requires the user to take a certain number of precautions:

- It is unnecessary and ineffective to have a checking instruction `/CONTROL/ RANDOM=...;` Indeed, each slave reads its random germ in the shared file `ranpar.dat`.
- **Watch out for the write-read instructions in files** (`WRITELN`, `WRITE`, `GET`, `GETLN`, `PRINT`, `SETTRACE:`, etcetera) during replication. The slaves do not write to the standard output of the master. If you use a file, you are advised to give its absolute path, as the

slaves are not necessarily started from the same directory as the master. These files must also be visible by NFS installation by all the stations used. You should also make sure that operations on a file (assign, read, write) are done by the same process (a “slave” can only write on a file if it is the slave - and not the master - which previously assigned and opened it).
For each PVM session, the standard output (PRINT, ...) of the slave processes is redirected to a file called `/tmp/pvml.*`

- You are strongly advised not to debug (`/CONTROL/ OPTION=DEBUG;`) a model which is using parallel replications.
- The CPU time taken functions (GETCPUT, GETPROFILE) only work on one process at a time.
- **Parallel replications and serial replications do not necessarily behave in the same way.** Parallel replications are started from the same simulation context, which is not necessarily the case for serial replications. For example, a replication may change a global variable of the model. In serial replications, the next replication will look at the changed value of that variable, while each parallel replication will use the initial value. Each user must be careful over this point if ever he wants to operate in parallel models designed for serial replications.
- You are strongly advised to run only one slave at one time on each station, so as to avoid collisions when information is transferred from the master to the slave and vice-versa.
- The slaves do not need a licence for use. So parallel replications can be run on several stations with only one licence.
- If an error occurs during replication, QNAP2 stops with a corresponding error message and the “slave” processes are also stopped. Example:

```
 *** SIMULATION WITH REPLICATION METHOD ***
   262147: ==>ERROR (INTER) : A REFERENCE WITH VALUE "NIL" IS USED
   262147:                    LINE NUMBER :     14
   262147: ... ACTIVE STATION : q2       FOR CUSTOMER :      3(        )
   262147: ... TIME =      1.842
   262147:                    LINE NUMBER :     14
 Terminated with errors: slaves being killed
 Please check that no zombie remains before
 launching a new parallel Qnap2
     25
     26 /END/
STOP: QNAP2 : END OF EXECUTION
```

Here, 262147 is the process number of the QNAP2 “master ”, given in decimal.

- If ever you cannot add a station to the list of stations that PVM can use:

```
pvm> add averell
0 successful
                    HOST     DTID
                 averell Can't start pvmd
```

You should check that all the operating conditions are met (directories and files seen by the station in question, environment variables properly placed, ...) and also that there are no (network) access problems to this station.

5.4.7.4 List of communication interface commands.

This paragraph summarizes the online help given (`help` command) by the `pvm` communication interface:

```
add    - Add hosts to virtual machine
Syntax:  add hostname ...

alias  - Define/list command aliases
Syntax:  alias [name command args ...]

conf   - List virtual machine configuration
Syntax:  conf
Output fields:
HOST    host name
DTID    tid base of pvmd
ARCH    xhost architecture
MTU     UDP max packet size
SPEED   host relative speed

delete - Delete hosts from virtual machine
Syntax:  delete hostname ...

echo   - Echo arguments
Syntax:  echo [ arg ... ]

halt   - Stop pvmds
Syntax:  halt

help   - Print helpful information about a command
Syntax:  help [ command ]

id     - Print console task id
Syntax:  id

jobs   - Display list of running jobs
Syntax:  jobs [ options ]
Options:  -l   give long listing

kill   - Terminate tasks
Syntax:  kill [ options ] tid ...
Options:  -c   kill children of tid

mstat  - Show status of hosts
Syntax:  mstat name ...

ps     - List tasks
Syntax:  ps [ -axh ]
Options:  -a       all hosts (default is local)
          -hhost   specific host tid
          -nhost   specific host name
          -x       show console task (default is not)
Output fields:
  HOST    host name
  A.OUT   executable name
  TID     task id
  PTID    parent task id
  FLAG    status
FLAG values:
  c   task connected to pvmd
  a   task waiting authorization
  o   task connection being closed

pstat  - Show status of tasks
Syntax:  pstat tid ...

quit   - Exit console
Syntax:  quit

reset  - Kill all tasks
Syntax:  reset

setenv - Display or set environment variables
Syntax:  setenv [ name [ value ] ]

sig    - Send signal to task
Syntax:  sig signum task ...

spawn  - Spawn task
Syntax:  spawn [ options ] file [ arg ... ]
Options:  -(count)  number of tasks, default is 1
          -(host)   spawn on host, default is any
          -(ARCH)   spawn on hosts of ARCH
          -?        enable debugging
          ->        redirect output of job to console
          ->(file)  redirect output of job to file
          ->>(file) append output of job to file

unalias - Undefine command alias
Syntax:  unalias name

version - Show libpvm version
Syntax:  version
```

5.5 Simulation results

5.5.1 Overview

Queue statistics are automatically computed on queue related parameters: number of customers, service time, response time, blocked time, utilization rate, throughput. Default queue statistics are computed for all queues in the network. The user can request that queue statistics be computed separately for specific customer classes. The user can also request that queue statistics be restricted to specific queues and/or specific parameters.

User statistics are computed on user variables declared as *watched* variables. A watched variable is an ordinary integer or real variable, except that its declaration is prefixed with the WATCHED keyword. It can be used as any other variable in the algorithmic language. Changes in the value of a WATCHED variable are automatically tracked by the simulator in order to produce standard statistics.

The basic statistics are the mean, minimum, maximum, and variance. Additional statistics are available upon request: confidence intervals, marginal probabilities (histograms) and correlation coefficients.

All statistics can be computed in two modes:

Global statistics are computed over the entire measurement session. They are used to characterize the steady state of the model.

Partial statistics are computed over the last measurement period, i.e., between the last two sampling instants defined by the SAMPLE or SETSTAT:SAMPLE procedures. They are used to analyze transient states.

All computed results can be obtained with the *result access functions* presented in section 5.5.3.2.

5.5.1.1 Queue statistics

The queue statistics are computed from the following quantities measured during the simulation run (these quantities cannot be accessed by the user).

- T_m total duration of the measurement session,
- $T(q, n)$ total duration during which station q contained exactly n customers,
- $T(q, k, n)$ total duration during which station q contained exactly n class k customers,
- $N(q)$ total number of customers having left the station q (at the end of a quantum a customer does not leave the station),
- $N(q, k)$ total number of class k customers having left the station q,
- $S(q)$ total sum of the services completed in station q,
- $S(q, k)$ total sum of the services completed in station q for class k customers,
- $S'(q)$ total sum of the services performed in station q (including the uncompleted services),
- $S'(q, k)$ total sum of the services performed in station q for class k customers (including the uncompleted services),
- $R(q)$ total sum of the time periods spent in station q by the customers,
- $R(q, k)$ total sum of the time periods spent in station q by class k customers.

mean service time: the mean service time is computed with the total number of customers having left the station. It is the mean time during which these customers have occupied a server of the station. The uncompleted services are not taken into account in the computation of this mean value.

- global: $S(q)/N(q)$
- class k: $S(q,k)/N(q,k)$

mean response time: the mean response time is computed with the total number of customers having left the station. It is the mean time they spent in the station (waiting or being served). The customers still in the station are not taken into account in the computation of this mean value.

- global: $R(q)/N(q)$
- class k: $R(q,k)/N(q,k)$

mean number of customers: The mean number of customers in the station includes the customers having left the station and the customers remaining in the station at the end of the measurement session.

- global: $(\sum_{i=1}^{\infty} iT(q,i))/T_m$
- class k: $(\sum_{i=1}^{\infty} iT(q,k,i))/T_m$

Note:

There is no distinction between customers waiting or being served.

mean blocked time: This is the mean time during which a customer is blocked (waiting for a resource, a semaphore or a flag). As for the mean service time calculation, the customers which have not left the station are ignored.

A customer is blocked when it is waiting for a resource, a semaphore, a flag or on a join condition.

Note:

- A blocked customer is in service, so it uses a server in its current queue.
- The BLOCK and UNBLOCK procedures do not block the customers, but the servers. The blocked time of the servers is not accounted to the customers.

utilization rate: the mean utilization rate is the fraction of servers (or of resource units) which have been used (i.e occupied by customers) during the measurement session.

- global: $(S'(q)/T_m)/M$
- class k: $(S'(q,k)/T_m)/M$

where M is the number of servers (or resource units) in the station.

In the case of a single server it is equal to the mean server busy fraction, and also to the fraction of time the station was occupied by at least one customer.

In the case of a simple queue (SERVER, MULTIPLE(0) or SEMAPHORE) this value is always null.

In the case of an infinite server (INFINITE) the result is always zero.

Note:

For a multiple server, the utilization rate multiplied by the number of servers in the station yields the mean number of occupied servers.

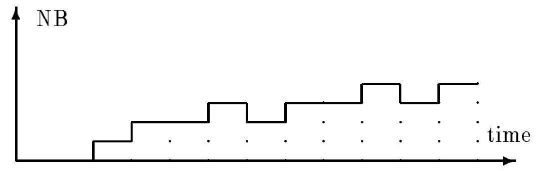
5.5.1.2 User statistics

User statistics are automatically computed on variables declared with the WATCHED keyword.

Watched variables (also called statistical variables) must be specified as *discrete* or *continuous* variables.

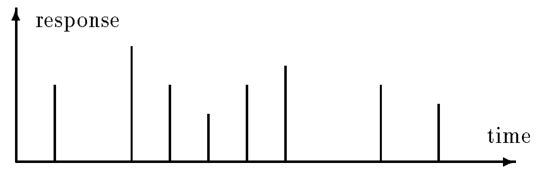
- A variable is said to be **continuous** when the computed statistics take time into account. Continuous variables are sometimes called *integrators.*

  The number of customers in a station is a *continuous* variable: the mean value makes sense only if we take into account the duration of the samples as show on the figure below.



- A variable is said to be *discrete* when the computed statistics do not take time into account.

  The response time of a queue is an example of discrete variable: the samples occur at specific instants, but the time spent between two samples is meaningless, as illustrated below.



QNAP2 can compute the same statistics on watched variables as on queue results. The statistics requests follow the same rules.

5.5.1.3 Statistics control

Syntax:

```
SETSTAT:{ ON | OFF } (stat_var_list);
/CONTROL/ TSTART = time;
```

Semantics:

All statistics can be individually controlled through the ON/OFF mechanism. The ON (resp. OFF) keyword is used to turn on (resp. off) statistics computation for `stat_var_list`, which stands for a list of observable objects:

- queues
- queue, class couples
- watched variables

The TSTART parameter of the /CONTROL/ command provides a global statistics control mechanism: the statistics computation of *all* variables starts at `time` and stops at the end of the simulation. TSTART is used primarily to suppress the startup period of the model, i.e., to start statistics only when the model has reached the steady-state.

The default value of the TSTART parameter is 0.

Note:

The ON/OFF mechanism overrides the TSTART parameter. TSTART is the default starting time for all statistics. The ON and OFF keywords explicitely start and stop statistics as soon as they are invoked.

Example:

```
/DECLARE/ WATCHED INTEGER SIZE;        & Size of I/O requests
          QUEUE DISK;

/CONTROL/ TMAX = 1000;
          TSTART = 100;

/EXEC/ BEGIN
          SETSTAT:DISCRETE (SIZE);
          SETSTAT:ON (SIZE);
          SETSTAT:QUEUE (DISK);
          SIMUL;
       END;
```

In this example, the statistics on the watched integer variable `SIZE` begin at `TIME = 0` whereas statistics on queue DISK begin only at `TIME = 100`.

5.5.1.4 Partial results

Syntax:

```
/CONTROL/ STATISTICS = PARTIAL | GLOBAL;
SETSTAT:PARTIAL (stat_var_list);
```

Semantics:

When the regeneration method is used, the STATISTICS parameter of the /CONTROL/ command, and the SETSTAT:PARTIAL procedure control the type of statistical results which are available at the end of each regeneration period.

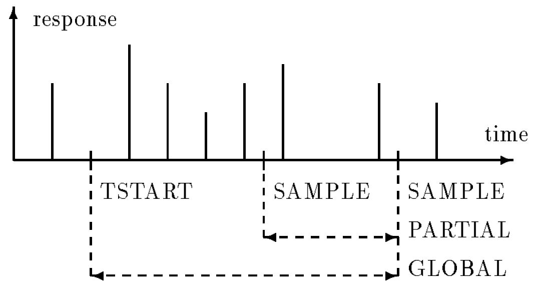
If the PARTIAL option is used, during a simulation run, a call to the OUTPUT procedure or to a result access function will produce the statistical results collected during the last regeneration period (i.e. during the interval of time between the last two calls to the SAMPLE or SETSTAT:SAMPLE procedures).

After the end of a simulation run, a call to the OUTPUT procedure or to a result access function will produce the statistical results collected during the complete simulation run (i.e. during the interval of time between the measurement starting time and the end of the simulation).

If the GLOBAL option is used (default option), then, during a simulation run, a call to the OUTPUT procedure or to a result access function will produce the statistical results collected between the measurement starting time and the last regeneration point (i.e. last call to the procedure SAMPLE or SETSTAT:SAMPLE).

After the end of a simulation run, a call to the OUTPUT procedure or to a result access function will produce the statistical results collected during the complete simulation run (i.e., between the measurement starting time and the end of the simulation).

The figure below illustrates the partial and global results concepts:



5.5.2 Statistics specification

The default statistics specifications are the following:

- Basic statistics (mean, min, max variance) are computed for all queues on all standard performance indicators (number of customers, service time, response time, blocked time, occupation rate).
- No queue statistics are computed per class.
- No statistics are computed for watched variables.

Note:

Using any SETSTAT:*keyword* procedure suppresses the default specifications. In this case, *only the requested statistics are computed.*

Two methods are provided to specify additional statistics or to suppress unneeded ones: SETSTAT:*keyword* procedures and /CONTROL/ parameters.

5.5.2.1 /CONTROL/ parameters

The following /CONTROL/ parameters can be used to request additional statistical results. Refer to chapter 3 on page 92 for details about these parameters.

Note:

These parameters are provided only for compatibility with previous QNAP2 releases. SETSTAT:*keywords* procedure calls always override /CONTROL/ parameters. It is not recommended to mix /CONTROL/ specifications and SETSTAT:*keyword* specifications, as the /CONTROL/ parameters will probably be ignored. Use of the SETSTAT:*keyword* procedures is encouraged as the algorithmic language is more powerful, more selective and more flexible than the command language.

CLASS is used to specify the queues on which class-specific results must be computed.

MARGINAL is used to specify the queues on which marginal probabilities on the number of customers must be computed.

ACCURACY is used to specify the queues and classes on which confidence intervals must be computed.

CORRELATION is used to specify the queues and classes on which correlation coefficients must be computed.

5.5.2.2 SETSTAT procedures

The following SETSTAT:*keyword* procedures are used to specify which statistics must be computed on queues or watched variables.

Basic queue statistics:

Syntax:

```
SETSTAT:QUEUE (queue_list);
SETSTAT:CLASS (queue_list, class_list);
```

Semantics:

QUEUE is used to request basic statistics on all standard results on the specified queues.

CLASS is used to request basic statistics on all standard results on the specified queues, with class-specific results for the specified classes.

Basic watched variable statistics:

Syntax:

```
SETSTAT:{ CONTINUE | DISCRETE } (watched_var_list);
```

Semantics:

CONTINUE and DISCRETE apply to watched variables only. They are used to specify whether the watched variables are *discrete* (e.g., mean size of requests) or *continuous* (e.g., average memory usage). See section 5.5.1.2, "Users statistics" above.

Using one of these keywords is an implicit request to compute basic statistics on the specified variables.

Additional statistics:

Syntax:

```
SETSTAT:ACCURACY (stat_var_list);
SETSTAT:PRECISION (stat_var_list, rel_error);
SETSTAT:CORRELATION (stat_var_list, order);
SETSTAT:MARGINAL (stat_var_list, intervals, start, width);
```

Semantics:

These keywords apply to all objects on which statistical results may be computed:

- queues
- queue, class couples

- watched variables

Note:

In order to request additional statistics, basic statistics must have been requested before:

- with the QUEUE and/or CLASS keywords (for queues and classes)
- with the CONTINUE or DISCRETE keyword (for watched variables)

ACCURACY is used to request the computation of confidence intervals.

PRECISION is used to specify the required accuracy. `rel_error` is the relative error. The result is considered as satisfactory when the ratio of the confidence interval to the estimated mean falls below `rel_error`. The simulation is stopped when *all* results are satisfactory. Note that confidence intervals must have been requested with the ACCURACY keyword.

CORRELATION is used within the regeneration method to request the computation of auto-correlation coefficients. `order` is the maximum order of the auto-correlation function. The default order is 5; the maximum allowed is 20. Note that confidence intervals must have been requested with the ACCURACY keyword.

MARGINAL is used to request the computation of marginal probabilities (histograms). `intervals` is the number of intervals; `start` is the lower bound of the first interval; `width` is the width of the intervals. QNAP2 computes the fraction of samples which fall within each interval, as well as those falling below the lower bound or above the upper bound.

Note:

When used on a queue or queue/class couple, these procedures apply to *all* queue standard results.

Restricted queue statistics:

Syntax:

```
SETSTAT:result:MEAN (queue_list [, class_list]);
SETSTAT:result:CORRELATION (queue_list [, class_list], order);
SETSTAT:result:MARGINAL (queue_list [, class_list],
                         intervals, start, width);
SETSTAT:result:ACCURACY (queue_list [, class_list]);
SETSTAT:result:PRECISION (queue_list [, class_list], rel_error);
```

where *result* is the name of a standard queue result:

BLOCKED for the blocked time of customers.

BUSYPCT for the utilization rate of the station.

CUSTNB for the number of customers in the station.

RESPONSE for the response time of the station.

SERVICE for the service time of the servers.

These procedures restrict statistics to specific results on the specified queues, with class-specific results on the specified classes.

The MEAN keyword is used to request the computation of basic statistics only (mean, min, max, variance). The use of the other keywords is exactly the same as explained above.

Miscellanous:

Syntax:

```
    SETSTAT:CANCEL (stat_var_list);
```

Semantics:
CANCEL is used to cancel any previous statistics specification on the given variables.

5.5.3 Statistical outputs

QNAP2 provides a standard statistics report at the end of the simulation. This report contains the standard results on each queue. Additional results on queues and watched variables are available through result access functions.

5.5.3.1 Standard report

The standard report is automatically produced at the end of a simulation run; it contains all the performance criteria characterizing the stations of the network globally (all classes together) or for each customer class.

`/CONTROL/ OPTION = NRESULT;` can be used to suppress printing of the standard report. Alternately, the OUTPUT procedure can be used to print the standard report on demand.

The standard report is a table with one line for each queue, showing the mean results all classes together.

When class-specific results are requested, the table contains one more line for each class. The name of the class is printed in the first column. The rest of the line shows the mean results for the class.

When confidence intervals are requested, the table contains one more line. The first column contains a "+/−" string. The rest of the line contains the confidence intervals.

NAME is the name of the queue or class, or the "+/−" string.

SERVICE is the service time.

BUSY PCT is the occupation rate.

CUST NB is the number of customers in the station.

RESPONSE is the response time.

SERV NB is the number of served customers: customers that actually have left the station.

At the end of a simulation run, it is also possible to obtain detailed information on the current state of the network (a complete list of the queue and customer attributes is given in the QNAP2 Reference Manual).

Note:

1. The final report does not include the stations from which no customer has left;
2. The Little formulae:

   `MCUSTNB` (*queue*) `= MRESPONSE` (*queue*) `* MTHRUPUT` (*queue*)

   and

   `MCUSTNB` (*queue, class*) `= MRESPONSE` (*queue, class*) `* MTHRUPUT` (*queue, class*)

   are not exactly verified at the end of a simulation run. This is a consequence of the definitions given above: in fact the mean number of customers in the station includes the customers remaining in the station while the mean response time does not. However, if the working state of the model is "stable" and if the simulation duration is long enough, the effect of the customers remaining in the station should be small and the formulae should be verified with a good approximation.

When marginal probabilities on the number of customers are requested with the MARGINAL parameter of the /CONTROL/ command, the standard report table is followed by a printout of the marginal probabilities, the variance of the number of customers and the variance of the response time.

5.5.3.2 Result access functions

During the simulation, the values returned by the result access functions depend on the partial/global option:

- When partial results have been requested with the `STATISTICS = PARTIAL` parameter of the /CONTROL/ command or the SETSTAT:PARTIAL procedure, the values returned by the result access functions apply to the previous sampling interval, i.e., the time interval between the last two SAMPLE or SETSTAT:SAMPLE procedure calls.
- When global results have been requested (default option), the values returned by result access functions apply to the time interval between the start of the measurements (as defined by the TSTART parameter of the /CONTROL/ command or by the SETSTAT:ON procedure) and the last call to SAMPLE or SETSTAT:SAMPLE.

GETSTAT functions for queues:

Syntax:

```
GETSTAT:result:MEAN (queue [, class])
GETSTAT:result:MINIMUM (queue [, class])
GETSTAT:result:MAXIMUM (queue [, class])
GETSTAT:result:VARIANCE (queue [, class])
GETSTAT:result:ACCURACY (queue [, class])
GETSTAT:result:CORRELATION (stat_var, n)
GETSTAT:result:MARGINAL (stat_var, n)
```

where `result` stands for the required result (SERVICE, RESPONSE, BLOCKED, CUSTNB, BUSYPCT)

MEAN, MAXIMUM, MINIMUM, and VARIANCE return the corresponding value.

ACCURACY returns the confidence interval.

CORRELATION returns the nth coefficient of the auto-correlation function.

MARGINAL returns the probability that the result falls within the nth interval, as specified with SETSTAT:MARGINAL.

Example:

```
/DECLARE/ QUEUE SYSTEM;

...

PRINT ("Response time of the system:",
       GETSTAT:RESPONSE:MEAN (SYSTEM),
       "+/-",
       GETSTAT:RESPONSE:ACCURACY (SYSTEM));
```

Queue throughput:

Syntax:

```
SERVNB (queue [, class])
queue.NBOUT [(class)]
queue.NBIN [(class)]
MTHRUPUT (queue [, class])
```

Semantics:

SERVNB and NBOUT both return the number of served customers (customers that have left the station).

NBIN returns the number of customers that entered the station, i.e. `NBOUT - NBIN` yields the number of customers that are still in the station.

MTHRUPUT returns the mean throughput, computed as the number of served customers divided by the observation period.

GETSTAT functions for watched variables:

Syntax:

```
GETSTAT:MEAN (stat_var)
GETSTAT:MAXIMUM (stat_var)
GETSTAT:MINIMUM (stat_var)
GETSTAT:VARIANCE (stat_var)
GETSTAT:ACCURACY (stat_var)
GETSTAT:CORRELATION (stat_var, n)
GETSTAT:MARGINAL (stat_var, n)
```

Semantics:

MEAN, MAXIMUM, MINIMUM and VARIANCE return the corresponding value.

ACCURACY returns the confidence interval.

CORRELATION returns the nth coefficient of the auto-correlation function.

MARGINAL returns the probability that the variable falls within the nth interval, as specified with SETSTAT:MARGINAL.

Example:

```
/DECLARE/ WATCHED INTEGER PKSIZE;

...

PRINT ("Packet sizes fall between",
       GETSTAT:MINIMUM (PKSIZE),
       " and ",
       GETSTAT:MINIMUM (PKSIZE));
```

Miscellanous:

The following result access functions are provided for compatibility with earlier QNAP2 releases. Their use is discouraged as the GETSTAT:*keyword* functions are more general.

- `MSERVICE (queue [, class])` mean service time
- `CSERVICE (queue [, class])` service time confidence interval
- `MBUSYPCT (queue [, class])` mean occupation rate
- `CBUSYPCT (queue [, class])` occupation rate confidence interval
- `MRESPONSE (queue [, class])` mean response time
- `CRESPONSE (queue [, class])` response time confidence interval
- `VRESPONSE (queue [, class])` variance of the response time
- `MBLOCKED (queue [, class])` mean blocked time
- `CBLOCKED (queue [, class])` blocked time confidence interval
- `MCUSTNB (queue [, class])` mean number of customers
- `CCUSTNB (queue [, class])` number of customers confidence interval
- `VCUSTNB (queue [, class])` variance of the number of customers
- `PCUSTNB (n, queue [, class])` marginal probability on the number of customers
- `MAXCUSTNB (queue [, class])` maximum number of customers
- `PMXCUSTNB (queue [, class])` maximum number of customers during the last sampling interval

5.5.4 Simulation process

The SIMUL procedure places on the time scale of the model under study the various events occuring in the model (enqueuing, preemption,...); it interprets the actions performed by these events and makes the appropriate modifications on the objects and variables of the model.

The simulation always follows the event order on the time scale of the model: no event can be processed if all the earlier events have not yet been handled. If several events occur at the same time, the simulator obviously handles sequentially actions that are supposed to be simultaneous. The actual order may be important for the behaviour of the model (for instance if two stations read and write simultaneously a shared variable). The order chosen by the model to handle simultaneous events should be taken into account.

At any instant of time, the model executes operations for a given process: a customer, a timer or an exception. This process is named the current process. Timers and exceptions become the current process only when their handler procedure is activated.

A customer may become the current process (we shall say "the current customer") if it verifies the following conditions:

- a server is allocated to this customer,
- it is not waiting (on a semaphore, resource, flag or join condition),
- it has to execute an operation at the present time (transition, work demand,...).

If several customers are candidates to be the current customer, the simulator selects first the customer with the highest priority level. If several candidates have the same priority level the simulator selects the one for which the operation was "planned" first.

When a customer has been elected as the current customer, the simulator performs all the operations associated with this customer until one of the following events occurs:

- the customer calls a work demand procedure (even if the value of this demand is null),
- end of a quantum,
- the customer enters a station where it cannot seize any server,
- the customer performs a P, WAIT, JOIN procedure on a semaphore or on a resource or on a flag, and this procedure is effective, i.e., blocks the customer,
- the customer performs a V, SET, FREE, TRANSIT or MOVE procedure activating a customer with a higher priority level,
- the customer performs a PRIOR procedure giving to another customer a higher priority than itself.

Note:

1. all the operations executed in a model (transitions, server allocations, procedure calls,...) are instantaneous on the time scale of the model. Only the work demands cause the current time to progress.
2. it should be noted that a customer remains "current customer" after a transition if it seizes a server in the station it enters (and if its priority level is not decreased).
3. a null work demand (for example CST(0)) may be used to force customers to give their turn to customers of the same priority level.

5.5.4.1 Tracing facilities

The discrete event simulator provides a full set of event tracing facilities. These tracing facilities are described in detail in chapter 9.

5.5.4.2 Model state information

The discrete event simulator maintains a large set of information during the simulation process. Part of these information is directly available from the predefined variables (e.g., TIME) and the object attributes (e.g., NB attribute of QUEUEs).

Other information is available via GETSIMUL:*keyword* functions.

Syntax:

```
GETSIMUL:FIRSTPROC
GETSIMUL:NEXTPROC (process)
GETSIMUL:WAKETIME (process)
GETSIMUL:PRSTATUS (process)
GETSIMUL:REPLINB
GETSIMUL:CURREPLI
```

Semantics:

FIRSTPROC returns a reference (REF ANY) to the first process in the event list. It can be a reference to a customer or a timer. It usually references the current process *except* when the current process is an exception. In this case, the first process is the customer or timer that waits to become the current process.

NEXTPROC returns a reference to the process following `process` in the event list. `process` must be a REF ANY pointing at a customer or a timer obtained with a previous call FIRSTPROC or NEXTPROC.

WAKETIME returns the expected wake-up time of the specified process. Note that this is meaningful only when this expected time is known, i.e., the process is either

- a customer performing a work demand procedure, or
- an armed timer.

When this is not the case, the returned value is -1.0.

PRSTATUS returns the status of the process as an integer code:

0 spleeping (i.e., idle timer or waiting customer)
1 waiting for its wake-up time
2 current process
3 customer blocked by a synchronization operation

REPLINB and CURREPLI are used with the replication method. **REPLINB** returns the number of replications. **CURREPLI** returns the number of the current replication.

Macro Processing $\boxed{6}$

6.1 Macro statement definition

Syntax:

macro_definition	→	`$MACRO` macro_id [(param_id [,...])]
		...
		`$END`
macro_id	→	identifier
param_id	→	identifier

Semantics:

This mechanism is intended to facilitate the writing of frequently used instruction sequences. This mechanism is part of the syntactic analyzer and therefore may be used to write algorithmic sequences and control language sequences.

A macro definition begins with a $MACRO statement specifying the identifier of the macro together with a list of dummy arguments and ends with a $END statement. The macro identifiers do not have to be declared. The dummy arguments may be used in the body of the macro definition to represent identifiers or constants. These identifiers are meaningless outside the macro statement and do not have to be declared.

- a macro definition may consist of any sequence of instructions. Its analysis is performed only when the macro is called. A macro may not contain calls or definitions of other macros.
- a macro definition may occur anywhere in a model, between two statements (parameters, algorithmic language statements,...).

The number of dummy arguments is limited to 20.

Note:

The semicolon is not a separator in the case of macros; it is not to be used to terminate the $MACRO statement, the $END statement, nor with the macro call.

6.2 Macro statement call

Syntax:

macro_call	→	$macro_id [(actual_param [,...])]
actual_param	→	identifier |
		[+ | −] integer_cst |
		[+ | −] real_cst |
		string_cst |
		boolean_cst

Semantics:

A call to a macro statement is equivalent to the copy of the whole macro definition body, in which the dummy arguments are replaced by the actual parameters into the QNAP2 source file, in place of the call.

The actual parameters must be simple syntactic tokens: expressions are not allowed. Procedure, function or type identifiers may be used as actual parameters of a macro.

A macro statement may be called anywhere in a model definition except within a macro definition.

Example:

```
/DECLARE/ QUEUE CPU, DA, DB;

$MACRO DISK (N, TETA)
/STATION/ NAME    = N;
          TRANSIT = CPU;
          SERVICE = EXP (TETA);
$END

& Macro calls:

$DISK (DA, 25)
$DISK (DB, 34)
```

Generated text:

```
/STATION/ NAME    = DA;
          TRANSIT = CPU;
          SERVICE = EXP (25);

/STATION/ NAME    = DB;
          TRANSIT = CPU;
          SERVICE = EXP (34);
```

Interface with Fortran $\boxed{7}$

QNAP2 can be connected to Fortran subroutines through the UTILITY feature. Note that a Fortran subroutine may call other subroutines written in any language. This enables to connect a QNAP2 model to any other software, depending on the compiler and loader features, which are operating-system dependent. The interface is defined in Fortran for conveniency.

The interface must be considered from the two aspects: QNAP2 side and Fortran side.

On the QNAP2 side resides the UTILITY procedure. This is a predefined QNAP2 procedure that can be called in any algorithmic language sequence. Calling this procedure makes QNAP2 call a Fortran subroutine called UTILIT.

The UTILIT subroutine is the entry point on the Fortran side. A default (empty) UTILIT subroutine is provided with QNAP2. The user may write his own UTILIT routine, replace the default one and link it with QNAP2. The UTILIT routine may in turn call other Fortran subroutines. When the UTILIT subroutine returns, the UTILITY procedure call is terminated.

Fortran subroutines are provided with QNAP2 in order to allow for communication between the model variables and the Fortran variables.

Note:

It is possible to call subroutines written in languages other than Fortran. This feature depends on the available compiler and linker. The main point is that the calling conventions be compatible with the calling conventions of the Fortran compiler.

It is also possible to tailor the Fortran UTILIT routine to handle the calling conventions and eventual convertion operations between Fortran and the other language.

7.1 UTILITY procedure

Syntax :

`UTILITY` (integer, array1 [, array2])

Semantics:

The UTILITY procedure is used to trigger from a QNAP2 program the controlled execution of an external user defined FORTRAN subroutine named UTILIT.

The integer is generally used as a code specifying what the UTILIT subroutine should do. The two arrays are used for data communication between the model and the UTILIT subroutine. The arrays must be INTEGER, REAL, BOOLEAN or STRING arrays.

7.2 UTILIT subroutine

Syntax:

```
      SUBROUTINE UTILIT (ICODE, IRTAB1, IRTAB2)
      INTEGER ICODE, IRTAB1, IRTAB2

C     user code

      RETURN
      END
```

Semantics:

The UTILIT subroutine is a Fortran routine. It must be declared with three dummy integer arguments. The body of the subroutine should not directly alter the values of the arguments. Communication with the model must be performed via special-purpose subroutines (see next section).

The user code may include named COMMON declarations. The blank common is reserved for QNAP2 internal use. The common names should begin with the letter U in order to avoid conflicts with QNAP2 commons.

7.3 Communication routines

Syntax:

```
QLOADI (IRTAB1 | IRTAB2, IQ1, IQ2, ITAB, IF1, IF2, IERR)
QLOADR (IRTAB1 | IRTAB2, IQ1, IQ2, RTAB, IF1, IF2, IERR)
QLOADB (IRTAB1 | IRTAB2, IQ1, IQ2, LTAB, IF1, IF2, IERR)
QLOADS (IRTAB1 | IRTAB2, IQ1, IQ2, CTAB, ITAB, IF1, IF2, IERR)
QSTORI (ITAB, IF1, IF2, IRTAB1 | IRTAB2, IQ1, IQ2, IERR)
QSTORR (RTAB, IF1, IF2, IRTAB1 | IRTAB2, IQ1, IQ2, IERR)
QSTORB (LTAB, IF1, IF2, IRTAB1 | IRTAB2, IQ1, IQ2, IERR)
QSTORS (CTAB, ITAB, IF1, IF2, IRTAB1 | IRTAB2, IQ1, IQ2, IERR)
```

Semantics:

These subroutines are provided for communication between the model and the Fortran subroutine. All communication is performed between the two QNAP2 arrays passed to UTILITY and user-defined Fortran arrays.

QLOADI is used to load an integer Fortran array with integer values stored in a QNAP2 array. The arguments are the following:

- IRTAB1 or IRTAB2 is used to specify which UTILITY array argument should be used.
- IQ1 (resp. IQ2) is used to indicate the index of the first (resp. last) element to read from the QNAP2 array.
- ITAB is a user-defined Fortran INTEGER array used to store the values.
- IF1 (resp. IF2) is used to indicate the index of the first (resp. last) element to store the data in the Fortran array.
- IERR is a status code returned by QLOADI. 0 means everything is ok. A positive value means something went wrong (e.g., bad array indices).

QLOADR and QLOADB are similar to QLOADI for real and boolean data. RTAB is a user-defined Fortran REAL array. LTAB is a user-defined Fortran LOGICAL array.

QLOADS is similar to QLOADI for string data. CTAB is a user-defined Fortran CHARACTER array. ITAB is a user-defined Fortran INTEGER array used to store the length of each character string. IERR takes the value 0 if everything is fine, 1 if a string is truncated, 2 if the operation failed.

QSTORI is used to store integer Fortran data into a QNAP2 array. The arguments are the following:

- ITAB is a user-defined Fortran INTEGER array.
- IF1 (resp. IF2) is used to indicate the index of the first (resp. last) element to read from the Fortran array.
- IRTAB1 or IRTAB2 is used to specify which UTILITY array argument should be used.
- IQ1 (resp. IQ2) is used to indicate the index of the first (resp. last) element to store the data in the QNAP2 array.
- IERR is a status code returned by QSTORI.

QSTORR, QSTORB and QSTORS are similar to QLOADI for real, boolean and string data.

In all cases, the QNAP2 array used should be of the appropriate type.

7.4 Example

We assume that a user needs in his QNAP2 model to sort values stored in an array and that he owns a good Fortran sorting routine. The principle is to use the `UTILITY` procedure in the QNAP2 program to transfer the array to the `UTILIT` subroutine where:

- the values are obtained from QNAP2,
- the sorting routine is called (named for instance `SORTIN`),
- the sorted values are sent back to QNAP2.

7.4.1 Qnap2 program

```
& ITAB: values to be sorted
& IRES: sorted values
/DECLARE INTEGER ITAB (10) = (3, 2, 7, 4, 10, 8, 5, 9, 1, 6);
         INTEGER IRES (10);
/EXEC/BEGIN
       UTILITY (1, ITAB, IRES);
       PRINT (IRES);
      END;
/END/
```

7.4.2 Fortran subroutine

```
C23456
      SUBROUTINE UTILIT (ICODE, IRTAB1, IRTAB2)
      INTEGER ICODE, IRTAB1, IRTAB2

C N:  array dimension
      INTEGER N
      PARAMETER (N=10)

C TABIN: values to be sorted

C TABOUT: sorted values

C IERR: error code
      INTEGER TABIN (N), TABOUT (N), IERR
      IF (ICODE.NE.1) GOTO 2

C QNAP2 --> FORTRAN
      CALL QLOADI (IRTAB1, 1, N, TABIN, 1, N, IERR)
      IF (IERR.GT.0) GOTO 1

C sorting (user routine)
      CALL SORTIN (N, TABIN, TABOUT)

C FORTRAN --> QNAP2
      CALL QSTORI (TABOUT, 1, N, IRTAB2, 1, N, IERR)
      IF (IERR.EQ.0) GOTO 2
C ERROR
```

```
 1 CONTINUE
       PRINT*,'ERROR'

 C END

 2 CONTINUE
   RETURN
   END
```

7.4.3 Link edition

In the link command, the routine `SORTIN` and the other user routines called by `SORTIN` must be specified.

An example for DEC VAX/VMS:

```
$ LINK /NOMAP /EXE=QNAP2 QNAP2.OBJ,QNAP2/LIB,BRIGITTE/LIB-
                    UTILIT.OBJ,SORTIN.OBJ
```

The link command resides in the QNAP2 installation command. Refer to the installation instructions for details.

The Qnap2 debugger $\boxed{8}$

8.1 Introduction

Whatever program is being operated, the first runs often fail to provide the expected results and then comes the difficult task of pin-pointing the source of error.
Usual methods in language-based programs consist of:

- Placing print-out statements for variables or diagnosis messages in the program code. This method is advantageous when the observation becomes significant over several iterations, as is the case for example with mathematical programs. A new executable has to be created to modify such orders, that is a recompilation of the relevant modules and a relisting of the links.
- Using an interactive debugger which, when available, is supplied either with the system or with the program. This chapter covers this type of tool.

The point at which program processing is broken off to enable the user to make demands of the debugger will be termed the break point.

A debugger permits the program process to be traced step by step, statement by statement, allows insertion of break points, consulting and modification of certain variable values, continuation of normal program process through to the next break point pre-specified by the user.

In the case of QNAP2, the debugger operates solely from the algorithmic code written by the user, it is supposed a priori that the declared entities are known. It follows that the debugger will be the most useful in simulation or when debugging /EXEC/ sequences.

This debugger was designed with the aim of minimizing the user learning curve which explains why it has been integrated into the QNAP2 algorithmic language. The Debugger functions are implemented in this language through a certain number of specific procedures.

8.2 The functions of debugger

This section describes the QNAP2 debugger functions and serves as a user manual. We will also see how to use the QNAP2 facilities to improve tool performance.

8.2.1 Available functions

The QNAP2 debugger is based on conventional debugging tool functions found in such languages as C, FORTRAN,...
The following operations are possible from first break point:

- place or remove other break points
- demand the value of a variable, whatever its nature
- modify a variable
- execute orders written in algorithmic language within quite broad limits. As the Debugger command language is integrated into QNAP2 algorithmic language, the demand for the value of a variable is expressed by an order such as PRINT and an executable sequence has to be in a BEGIN ... END block.

8.2.2 Running the debugging tool

This section describes how to use the QNAP2 debugger.

8.2.2.1 When constructing a model

It is essential that the relevant sections are compiled with the option:

```
/CONTROL/ OPTION=DEBUG;
```

This option is cancelled by

```
/CONTROL/ OPTION=NDEBUG;
```

A first break point must be placed in the QNAP2 program using the HALT procedure. This first break point is fixed and is no longer removable. When it is reached, a message giving the line number appears on the screen and the user can then enter any request in QNAP2 algorithmic language.

8.2.2.2 Debugger commands

Syntax:

```
HALT;
GO;
BREAK (line_num [, ...]);
SHBREAK;
CANCELBR (line_num [, ...]);
INSERT ([file] [, line_num [, ...]]);
REMOVE (line_num [, ...]);
SHINSERT;
STP;
```

HALT Generates a break point at the current line. If it is placed in an algorithmic sequence of the model it cannot be removed.

GO Enables restart of the normal QNAP2 program process until the next break point is reached.

BREAK Allows break points to be placed before the specified line numbers. The line numbers are those given in the FSYSOUTP output file when the model is compiled.

SHBREAK Produces a listing of all lines in which breakpoints has been inserted.

CANCELBR Enables removal of break points placed by BREAK on the specified lines. If no BREAK has been requested on such lines a message is printed.

INSERT Allows code to be inserted before processing the statements of the specified line. The code is read from the file if it is specified, otherwise it is read from the keyboard. If no line number is given the code is processed at the level of the current line. Statements cannot be inserted into lines which already have an INSERT or a BREAK statement.

REMOVE Enables removal of the code inserted on the specified lines with INSERT.

SHINSERT Prints out a list of lines into which INSERT blocks have been placed.

STP Allows simulation to be traced step by step. In the case where a customer is transited from one station to another, STP will show the statements of the following customer whose service has been compiled with the debugging option.

8.3 Example

This section illustrates the functions described above.

The facilities specific to the debugger are presented as are the predefined mechanisms of QNAP2 such as the save functions and library restoration. We will also see hows events are related to each other. We have used the file audition facilities to enable requests submitted to the debugger to be shown here.

8.3.1 Model declaration

```
SIMULOG *** QNAP2 *** ( 01-04-95  )  V9.2
(C) COPYRIGHT BY CII HONEYWELL BULL AND INRIA, 1986

/CONTROL/ OPTION= DEBUG; & Compilation under debugger

/DECLARE/ QUEUE A,B;
          INTEGER I;     & number of customers on station A
          FILE F1;       & auditor file

/STATION/ NAME=A;
          SERVICE=BEGIN
                    EXP(3.0);
                    I:=I-1;
                  END;
          TRANSIT=B;
          INIT=3;

/STATION/NAME=B;
          SERVICE=BEGIN
                    EXP(2.0);
                    I:=I+1;
                  END;
          TRANSIT=A;

/CONTROL/TMAX=20.0;

/EXEC/BEGIN
         FILASSIGN(F1,"session.lis");
         OPEN(F1,2);
         AUDIT(FSYSOUTP,F1," Debugger: ");
         AUDIT(FSYSTERM,F1," User request: ");
       END;

/EXEC/BEGIN
         I:=3;
         HALT;    & at least one break point
         SIMUL;
       END;
/END/
```

8.3.2 First interactive break point

The first break point is placed on line 33. The debugger is called as soon as QNAP2 executes this statement. The first break point cannot be removed, so it should be placed appropriately.

The process is started, the model has been compiled. During interpretation of the algorithmic code contained in the /EXEC/ block, the HALT statement is reached and the first break point is generated.

```
Debugger : HALT ON BREAK POINT BEFORE LINE : 33
User request : BREAK (9); & request for stop on line 9
User request : GO;
```

The difference between BREAK and HALT is that break points placed with BREAK can be removed, which is not the case with HALT.

8.3.3 Beginning of simulation

```
Debugger : *** SIMULATION ***
Debugger : HALT ON BREAK POINT BEFORE LINE : 9
User request : SHBREAK; & Requests list of break points
Debugger : EXISTING BREAK POINT BEFORE LINE : 9
User request : PRINT (I); & requests value of I
Debugger : 3
```

8.3.4 Using the environment SAVE and RESTORE functions

Here we deal only with the environment SAVE function. RESTORE will be handled later.

This function is used to detect a problem which arises at a late stage of the simulation. It avoids a costly and pointless interactive run.

```
User request : INSERT (11); & insertion of code on line 11
User request : IF (I=2) THEN SAVERUN ("MODEL");
User request : GO;

Debugger : (0B0H09) MODEL SAVED ON FILE FSYLIB : MODEL
```

The model is saved in the state where it was when SAVERUN was called. As we shall see, during retrieval, the break points and the inserted blocks are also saved.

8.3.5 Removal of an inserted block

```
Debugger : SHINSERT
User request : INSERT ON LINE : 11
User request : REMOVE (11);     & removal of block inserted in 11
User request : GO;
```

8.3.6 Step by step execution

```
Debugger : HALT ON BREAK POINT BEFORE LINE : 9
User request : STP; & single line step
Debugger : STEP BEFORE LINE : 18
```

Note that with the station A customer being blocked by the time constant, the station B customer becomes the current customer and processing continues on line 18.

```
User request : STP;
Debugger : STEP BEFORE LINE : 19
User request : STP;
Debugger : STEP BEFORE LINE : 10
```

The customer at station B is in turn blocked by a time delay. Once again, customer A becomes the current customer.

```
User request : STP;
Debugger : STEP BEFORE LINE : 11
```

8.3.7 Cancelling a specified break point

```
User request : CANCELBR (9);     & removal of break point on 9
User request : GO;
```

As no further break point is defined, the simulation continues up to its normal end.

8.3.8 Restoring the context saved during debugging

We restart a session by restoring the simulation from its previously saved state. This is done with the command shown below (see section 2.6.2 for explanations about saving and restoring models).

```
      /EXEC/ RESTORE;
```

After restoring the model, the execution is resumed. Note that the model run after retrieval is exactly the same as the previous run.

```
Debugger : MODEL RESTORED : MODEL
Debugger : HALT ON BREAK POINT BEFORE LINE : 9
User request : SHBREAK;
Debugger : EXISTING BREAK POINT BEFORE LINE : 9
User request : SHINSERT;
Debugger : INSERT ON LINE : 11
```

The break and insertion points are the same.

```
User request : REMOVE (11);
User request : STP;
Debugger : STEP BEFORE LINE : 18
User request : STP;
Debugger : STEP BEFORE LINE : 19
User request : STP;
Debugger : STEP BEFORE LINE : 10
User request : STP;
Debugger : STEP BEFORE LINE : 11
User request : STP;
Debugger : STEP BEFORE LINE : 17
```

Trace Facilities $\boxed{9}$

This chapter presents the trace facilities available in simulation. Tracing facilities available with the analytical and Markovian solvers, are decribed in chapter 5 with each solver.

QNAP2 provides a full set of tracing facilities with the discrete event simulator. The standard trace prints all the events occuring in the model in a general purpose format. The generalized trace allows the user to specify which events must be traced and how to process them.

9.1 Trace control

Syntax:

```
SETTRACE:ON;
SETTRACE:OFF;
SETTRACE:BOUNDS (start, end)
SETTRACE:WIDTH (80 | 132);
SETTRACE:BRIEF;
SETTRACE:LONG;
```

Semantics:

ON and OFF are used to enable and disable tracing immediately. They can be used in any algorithmic code sequence.

BOUNDS is used to enable tracing within a given time period. The two parameters represent the starting and ending times of the event trace.

WIDTH controls the width of the standard trace listing. Only 80 and 132 are allowed, corresponding to two different presentation formats.

BRIEF is used to select only the event trace messages (default option). **LONG** adds the line number of the source code corresponding to each traced event.

The following parameters of the /CONTROL/ command can also be used to control the trace. Refer to section 3.8, "/CONTROL/ Command" for details.

```
TRACE = [ start [, end ] ] [, width] [, BRIEF | LONG ];
OPTION = [N]TRACE;
UNIT = TRACE (file);
```

9.2 Standard Trace

The standard event trace contains the following events together with their date of occurence:

- initialization of customers,
- work demands and operations performed by customers in the servers,
- transitions of customers,
- expiration of timers.

The trace information is presented in a column form. The column contents are the following:

col 1: customer identification

col 2: queue containing the customer

col 3: class of the customer

col 4: priority of the customer

col 5: number of customers in the queue

col 6: operation performed by the customer

At each time modification, the TIME value is printed. Each traced event is preceded by the character ">" in the first column.

With the LONG option, the line number of the source code corresponding to each traced event is printed on a separate line.

Example:

```
SIMULOG   ***  QNAP2  ***  ( 01-04-95  ) V 9.2
(C)  COPYRIGHT BY CII HONEYWELL BULL AND INRIA, 1986


    1             /DECLARE/ QUEUE A,B,C,R;
    2                       INTEGER N;
    3                       CLASS X,Y;
    4
    5             /STATION/ NAME     = A;
    6                       SCHED    = PRIOR,PREEMPT;
    7                       INIT     = N;
    8                       PRIOR(X) = 1;
    9                       PRIOR(Y) = 2;
   10                       TRANSIT  = B,1,C,2;
   11                       SERVICE  = EXP(1.);
   12
   13             /STATION/ NAME = R;
   14                       TYPE = RESOURCE;
   15
   16             /STATION/ NAME    = B;
   17                       TRANSIT = A;
   18                       SERVICE = BEGIN
   19                                   P(R);
   20                                   EXP(3.);
   21                                 END;
   22
```

```
    23              /STATION/ NAME = C;
    24                        COPY = B;
    25
    26              /CONTROL/ TMAX = 100.;
    27                        TRACE = 0., 2.;
    28                        OPTION = NRESULT;
    29
    30              /EXEC/    BEGIN
    31                          N:=2;
    32                          SIMUL;
    33                        END;
*** INITIALISATION ***
>      1  A         X            1     4 INITIAL CREATION AT QUEUE A
>      2  A         X            1     4 INITIAL CREATION AT QUEUE A
>      3  A         Y            2     4 INITIAL CREATION AT QUEUE A
>      4  A         Y            2     4 INITIAL CREATION AT QUEUE A
-------------------------------------------------------------------------------
 CUSTOMER   QUEUE   CLASS   PRIOR    NB            OPERATION
-------------------------------------------------------------------------------

TIME: 0.000
> TIMER TSYSTRAC                          IS JUST ACTIVATED
>      3  A         Y            2     4   DELAY       0.901 UNTIL        0.901

TIME: 0.901
>      3  A         Y            2     3   ==>
          C         Y            2     1
>      3  C         Y            2     1   P R        VALUE:      0
>      3  C         Y            2     1   DELAY       8.197 UNTIL        9.097
>      4  A         Y            2     3   DELAY       0.879 UNTIL        1.780

TIME: 1.780
>      4  A         Y            2     2   ==>
          B         Y            2     1
>      4  B         Y            2     1   P R        VALUE:      0
>      1  A         X            1     2   DELAY       0.027 UNTIL        1.807

TIME: 1.807
>      1  A         X            1     1   ==>
          C         X            1     2
>      2  A         X            1     1   DELAY       0.416 UNTIL        2.223

TIME: 2.000
> TIMER TSYSTRAC                          IS JUST ACTIVATED
    34              /END/
```

9.3 Generalized Trace

The generalized tracing facilities allow the user to specify which events to trace and what tracing actions to perform. The tracing actions are completely general, as they are specified directly in the algorithmic language. For example, the standard trace can be completely emulated with the generalized trace.

The generalized tracing facilities are designed not only for model debugging, but also for tailored applications: user trace, animation trace.[1] It can also be used to catch special events and process them as needed.

9.3.1 Overview

The generalized trace provides the following features:

- Restrict tracing to specific objects: queues, classes, customers, timers, flags.
- Restrict tracing to specific events: work demand procedures, synchronization operations.
- Generalized tracing actions: the tracing actions are not restricted to printouts. The user specifies a trace procedure to trigger for each traced object/event.

User-defined tracing procedures may not have parameters. References on parameterless generic procedures are also allowed. The actual procedure called depends on the value of the reference when the traced event occurs.

A user-defined tracing procedure may contain any algorithmic code, except simulation operations: a tracing procedure may not cause traceable events.

Predefined procedures are available to output the standard trace messages. Predefined functions are available to obtain information about the event that triggered the procedure, and the related parameters.

Specification of tracing actions is performed through SETTRACE:*keyword* procedures. Information about the traced event is obtained via GETTRACE:*keyword* functions.

9.3.2 Basic concepts

The following objects can be used to select the traced events:

1. Queues. A trace specification for a queue is considered as a default trace specification for all classes.
2. Customer classes
3. Customers.
4. Flags
5. Timers

When an event occurs during simulation, the actual tracing action performed is selected from the user specifications or from the default specifications. It is clear that several tracing actions can be candidates for the same event. For example, when a customer in queue A forces a customer in queue B to WAIT on flag F, several actions are candidates: those specified with queue A, with queue B, with WAIT events, and with flag F. The selection rules are based on three levels:

1. **Object role:** causing the event, undergoing the event or partner of the event.
2. **Priority** of the tracing actions as specified by the user.
3. **Default priority** of the tracing actions implemented by QNAP2.

The first selection criterion is the object.

[1]The animation features of MODLINE are built on top of the generalized trace.

1. When two trace specifications are candidates, QNAP2 selects the one with the highest priority level. If the priority levels are equal, QNAP2 selects the object that *undergoes* the event first, then the one that *causes* it, and finally the one that is a partner of the event.
2. If these criteria are not sufficient, QNAP2 selects a CUSTOMER or a TIMER first, then a FLAG or a QUEUE/CLASS couple, then a QUEUE, and finally a CLASS.
3. A tracing action specified as the default action never prevails over a user specification.

After selecting which object's tracing action is performed, QNAP2 considers whether a specific action depending on the event type has been defined. If no tracing action is defined for the occuring event, a default tracing action is performed.

This means that tracing specifications based on event types are *local* to objects. For example, a trace specification defined for flag F and WAIT events is valid only for flag F. It must be repeated for all flags if the user wants to catch all WAIT events.

9.3.3 Trace specifications

SETTRACE:*keyword* procedures are used to specify tracing actions, to enable/disable tracing and to produce the standard trace messages.

Syntax:

```
SETTRACE:SET (object ... [, event ... ] [, procedure ] [, priorities ]);
```

Semantics:

The SET keyword is used to assign a tracing action and priority levels to objects and events.

object is an object or a list of objects. A trace specification applies to queues, customer classes, timers and flags. It also applies to queue/class pairs, as in:

```
/DECLARE/ QUEUE cpu;
          CLASS batch;

          PROCEDURE ENDBATCH;
          BEGIN
            PRINT (TIME, "End of batch job");
          END;
          ...

          SETTRACE:SET (cpu, batch, "ENDSERV", ENDBATCH);
          ...
```

The *event*s to trace are specified as character strings containing symbolic event names. The symbolic names of the events are listed below.

procedure is either the identifier of a procedure without parameter, or a reference to a generic procedure without parameter. The constant NIL is allowed, which suppresses any previously specified tracing action. This is the most straightforward method to turn off the standard trace.

Example:

The following code turn offs the standard trace for all events except those related to queue A:

```
    /DECLARE/ QUEUE A;

              PROCEDURE TRACEIT;
              BEGIN
                SETTRACE:DISPLAY;
              END;

    /EXEC/ BEGIN
              SETTRACE:DEFSET (NIL);
              SETTRACE:SET (A, TRACEIT);
           END;
```

priorities is a list of one to three integers specifying the prority level of the trace specification depending on the role of the object. The first priotity level is used when the object *causes* the event; the second one when it *undergoes* it; the third one when it is a partner of the event.

9.3.3.1 Symbolic event names

- **AFTCUST, BEFCUST, BLOCK, UNBLOCK, MOVE, TRANSIT, P, V, PMULT, VMULT, PRIOR, RESET, SET, WAIT, WAITAND, WAITOR:** see the corresponding procedures
- **FISSION, FUSION, MATCH, SPLIT:** see the corresponding parameters of the /STATION/ command
- **JOINALL:** JOIN on all sons
- **JOINEND:** end of a JOIN
- **JOINLIST:** JOIN on an explicit list of sons
- **JOINNB:** JOIN on a specific number of sons
- **ENDSERV:** end of service and departure of the customer. This is the normal completion of the SERVICE algorithmic code. The customer leaves the queue through the TRANSIT parameter of the /STATION/ command, *not* through the TRANSIT procedure.
- **ENDSTART:** end of the statistics starting period (TSTART parameter of the /CONTROL/ command)
- **FREE:** `FREE (customer)` with no other argument
- **FREEFLAG:** `FREE (customer, flag)`
- **FREEP:** `FREE (customer, queue)` free from a blocking semaphore request
- **INIT:** initial customer creation (INIT parameter of the /STATION/ command)
- **NEWCUST:** dynamic customer creation with the NEW function
- **SOURCE:** customer creation in a SOURCE station, through the regular source mechanism.
- **SERVTIME:** begin of a work demand procedure (delay)
- **TMRCANCL:** see SETTIMER:CANCEL
- **TMRSETTM:** TIMER wake-up time setting
- **TMRWAKUP:** TIMER wake-up (expiration)

The symbolic name ALLEV refers to all events.

9.3.3.2 Default tracing actions

Syntax:

```
SETTRACE:RESET (object [, event ... ]);
SETTRACE:DEFSET ([event, ...] procedure);
SETTRACE:DEFRESET [(event ...)];
SETTRACE:DISPLAY;
```

Semantics:

The RESET keyword is used to cancel a previous trace specification on a given object and restore the default tracing action.

The DEFSET keyword is used to specify a default tracing specification, to be applied when no specific tracing action has been specified for a given object.

The DEFRESET keyword is used to cancel a previous default trace specification and restore the standard tracing action.

The DISPLAY keyword is used to print the standard trace message. It can be used only inside a user-defined tracing procedure.

9.3.4 GETTRACE functions

The GETTRACE:*keyword* functions can be used inside trace procedures to obtain information about the traced events. The available keywords are briefly listed below.

Note:

1. The trace procedure is always executed *after* the traced event.
2. The available information depends on the traced event. Only EVCODE, CODENAME, NAMECODE and EVSTATUS always return a value (possibly meaningless for EVSTATUS).
3. A request to unavailable information is an error and stops the simulation.
4. The exact meaning of each keyword with respect to the actual event is listed in the Reference Manual with each event symbolic name.

EVCODE returns the numeric code of the event as an integer.

> **Note:**
>
> The values of these numeric codes are likely to change in future QNAP2 releases. It is better to rely on the symbolic event names.

NAMECODE and CODENAME are used to convert event codes (INTEGERs) respectively to and from symbolic names (STRINGs)

EVSTATUS returns the status of the event (e.g., whether a WAIT was blocking or not).

DELAY returns the actual delay requested by a work demand procedure. It may be used only for work demand events (SERVTIME).

DISTRI returns the numeric code of the work demand procedure. It may be used only for work demand events (SERVTIME).

PARDISTR (n) returns the nth parameter of the work demand procedure. It may be used only for work demand events (SERVTIME).

CCLASS and CPRIOR return respectively a reference to a class and an integer (e.g., the previous class and priority level of the customer undergoing a forced TRANSIT to a new class with a new priority level).

CPROVOKE and QPROVOKE return a reference to the customer that caused the operation and a reference to the queue that "caused" the operation (usually the queue containing that customer).

CSUBJECT and QSUBJECT return a reference to the customer that underwent the operation and a reference to the queue that "underwent" the operation (usually the queue containing that customer). These keywords may be used only for events involving exactly one subject customer or queue.

CSECONDR and QSECONDR return a reference to the customer that was a partner in the operation and a reference to the queue that "was a partner" in the operation (usually the queue containing that customer). These keywords may be used only for events involving exactly one partner customer or queue.

LCLASSNB returns the number of classes involved in the event. It may be used only with the PMULT event.

CLLISTGET (n) returns the nth class in the list.

LCUSTNB returns the number of customers involved in the event. It may be used only when the event involves a list of customers (JOIN, JOINC).

CLISTGET (n) returns the nth customer in the list.

LNUMNB returns the number of integers involved in the event. It may be used only with the PMULT or VMULT events.

NUMLISTGET (n) returns the value of the nth integer in the list.

LPRIONB returns the number of priorities involved in the event. It may be used only with the PMULT event.

PRILISTGET (n) returns the value of the nth priority in the list.

LQUNB returns the number of queues involved in the event. It may be used only when the event involves a list of queues (BLOCK, UNBLOCK, PMULT, VMULT).

QLISTGET (n) returns the nth queue in the list.

FLAG returns a reference to the flag involved in the event. This keyword may be used only for events involving one flag (SET, RESET, WAIT, FREE from a single flag).

LFLAGNB returns the number of flags involved in the event. It may be used only when the event involves a list of flags (WAITOR, WAITAND).

FLISTGET (n) returns the nth flag in the list.

WHICHPROVOKE returns 1 if the event was caused by a customer, 2 if is was caused by an exception, and 3 when it was caused by a timer.

EXCEPTPROVOKE returns a reference to the EXCEPTION object when the event is related to an exception.

TIMERPROVOKE returns a reference to the TIMER object that caused the event.

TIMERSUBJECT returns a reference to the timer that underwent the event.

NUMBER returns an integer argument related to the event (e.g., number of sons involved in a `JOIN (n)`)

9.3.5 Example

```
SIMULOG   ***  QNAP2  ***  ( 01-04-95  ) V 9.2
(C)  COPYRIGHT BY CII HONEYWELL BULL AND INRIA, 1986


/DECLARE/ QUEUE SRC, ROBOT1, ROBOT2, TOOL;
          CLASS LARGE, SMALL;

/STATION/ NAME = SRC;
          TYPE = SOURCE;
          SERVICE = CST (1);
          TRANSIT = ROBOT1, SMALL, 0.6, ROBOT2, LARGE, 0.4;

/STATION/ NAME = ROBOT1;
          SERVICE = BEGIN
                      P (TOOL);
                      EXP (1.5);
                      V (TOOL);
                    END;
          TRANSIT = OUT;

/STATION/ NAME = ROBOT2;
          SERVICE = BEGIN
                      P (TOOL);
                      EXP (2.0);
                      V (TOOL);
                    END;
          TRANSIT = OUT;

/STATION/ NAME = TOOL;
          TYPE = RESOURCE;

/DECLARE/

STRING STATMESS (0:2) = ("passed", "blocked", "rejected");

PROCEDURE RESUSE;                    & Trace resource usage
BOOLEAN REQUEST;
BEGIN
  REQUEST := (GETTRACE:EVCODE = GETTRACE:CODENAME ("P"));
  PRINT (TIME, " : ",
         GETTRACE:CPROVOKE, " in ", GETTRACE:QPROVOKE,
         IF REQUEST
            THEN " requested "
            ELSE " released  ",
         GETTRACE:QSUBJECT,
         IF REQUEST
            THEN " : " // STATMESS (GETTRACE:EVSTATUS)
            ELSE "");
END;

/EXEC/ BEGIN
         SETTRACE:DEFSET (NIL);          & Turn off std trace
         SETTRACE:SET (TOOL, RESUSE);
       END;


/CONTROL/ TMAX = 20;
          OPTION = TRACE, NRESULT;

/EXEC/ SIMUL;
```

```
1.000     :         1  in ROBOT1     requested TOOL       : passed
2.514     :         1  in ROBOT1     released  TOOL
2.514     :         2  in ROBOT1     requested TOOL       : passed
3.833     :         2  in ROBOT1     released  TOOL
3.833     :         3  in ROBOT1     requested TOOL       : passed
3.873     :         3  in ROBOT1     released  TOOL
4.000     :         4  in ROBOT2     requested TOOL       : passed
4.833     :         4  in ROBOT2     released  TOOL
5.000     :         5  in ROBOT2     requested TOOL       : passed
6.532     :         5  in ROBOT2     released  TOOL
6.532     :         6  in ROBOT2     requested TOOL       : passed
7.000     :         7  in ROBOT1     requested TOOL       : blocked
9.191     :         6  in ROBOT2     released  TOOL
9.702     :         7  in ROBOT1     released  TOOL
9.702     :         8  in ROBOT1     requested TOOL       : passed
9.917     :         8  in ROBOT1     released  TOOL
9.917     :         9  in ROBOT1     requested TOOL       : passed
10.37     :         9  in ROBOT1     released  TOOL
10.37     :        10  in ROBOT1     requested TOOL       : passed
10.44     :        10  in ROBOT1     released  TOOL
11.00     :        11  in ROBOT2     requested TOOL       : passed
14.00     :        14  in ROBOT1     requested TOOL       : blocked
15.59     :        11  in ROBOT2     released  TOOL
15.59     :        12  in ROBOT2     requested TOOL       : blocked
16.48     :        14  in ROBOT1     released  TOOL
16.48     :        15  in ROBOT1     requested TOOL       : blocked
18.21     :        12  in ROBOT2     released  TOOL
18.21     :        13  in ROBOT2     requested TOOL       : blocked
18.44     :        15  in ROBOT1     released  TOOL
18.44     :        16  in ROBOT1     requested TOOL       : blocked
```

```
  18.67     :        13  in ROBOT2     released  TOOL
  18.67     :        17  in ROBOT2     requested TOOL       : blocked
   56           /END/
```

Graphics and Qnap2 $\boxed{10}$

Until the V9.2 release, QNAP2 had included a few graphic commands that allowed to draw curves, bar charts and pie charts. Since QNAP2 V9.3, these commands (that are listed below) are no more available.

The graphic commands that have been suppressed are:

- CLEARSCR - clear of graphical window
- MBEGIN - initialization of graphical parameters
- MENDGR - end of graphical operations
- MODIFY - modification of graphical set up
- PLOATT - graphical set up of curves, bar charts and pie charts
- PLOCUR - curve plotting
- PLOHIS - bar chart plotting
- PLOSEC - pie chart plotting

They were suppressed because QNAP2 main purpose is not to perform graphical operations and, above all, because these graphic mechanisms were too rigid.

In the following pages, you will find examples of generation of *GNUPLOT* files for curves and bar charts plotting. *GNUPLOT* is a powerful chart generation and display tool, which is included in the *Modline* product.

These examples are integrated to the QNAP2 examples files.

10.1 Procedure GPLOCUR

This procedure is an example of GNUPLOT files generation procedure that can replace the old `PLOCUR` QNAP2 procedure, for curves plotting.

Procedure code:

```
/DECLARE/
FILE pc_datafile;
FILE pc_gnuplotfile;

PROCEDURE GPLOCUR (filename, nb_curves, nb_points, x, y, title, legend);

& Arguments:
& ----------

STRING filename;                                 & GNUPLOT file name
INTEGER nb_curves,                               & number of curves
        nb_points;                               & number of points
REAL x (nb_curves, nb_points),                   & abscissae
     y (nb_curves, nb_points);                   & ordinates
STRING title,                                    & chart title
       legend (nb_curves);                       & legends

& Preconditions (must be checked by the caller):
& -------------
& filename must be a legal file name
& (nb_curves >= 1)
& (nb_points >= 1)

& Local variables:
& ----------------

INTEGER icurve, ipoint;
STRING pref_datafile, ext_datafile, name_datafile;

BEGIN

  pref_datafile := "pc_";                        & data file name prefix
  ext_datafile  := ".dat";                       & data file name extension

  FILASSIGN (pc_gnuplotfile, filename);
  OPEN (pc_gnuplotfile, 3);

  WRITELN (pc_gnuplotfile, "set title '", title, "'");
  WRITELN (pc_gnuplotfile, "set autoscale");
  WRITELN (pc_gnuplotfile, "plot \");

  FOR icurve := 1 STEP 1 UNTIL nb_curves DO BEGIN
```

```
      name_datafile := CONVERT(icurve, STRING) // ext_datafile;
      FILASSIGN (pc_datafile, name_datafile);
      OPEN (pc_datafile, 3);

      FOR ipoint := 1 STEP 1 UNTIL nb_points DO BEGIN
          WRITELN (pc_datafile, x (icurve, ipoint), " ", y (icurve, ipoint));
      END;  & for

      CLOSE (pc_datafile);

      WRITE (pc_gnuplotfile, "'", name_datafile, "'");
      WRITE (pc_gnuplotfile, " title '", legend (icurve), "'");
      WRITE (pc_gnuplotfile, " with linespoints");

      IF (icurve < nb_curves) THEN WRITE (pc_gnuplotfile, ", \");
      WRITELN (pc_gnuplotfile);

  END;  & for

  WRITELN (pc_gnuplotfile, "pause -1 'Press return to continue...'");
  CLOSE (pc_gnuplotfile);

END;  & proc
```

10.2 Procedure GPLOBAR

This procedure is an example of GNUPLOT files generation procedure that can replace the old PLOHIS QNAP2 procedure, for bar charts plotting.

Procedure code:

```
/DECLARE/
FILE ph_datafile;
FILE ph_gnuplotfile;

PROCEDURE GPLOBAR (filename, nb_bars, nb_subparts, x, title, legend);

& Arguments:
& ----------

STRING filename;                                 & GNUPLOT file name
INTEGER nb_bars,                                 & number of bars
        nb_subparts;                             & number of bar parts
REAL x (nb_bars, nb_subparts);                   & values
STRING title,                                    & chart title
       legend (nb_bars);                         & legends

& Preconditions (must be checked by the caller):
& -------------
& filename must be a legal file name
& (nb_bars >= 1)
& (nb_subparts >= 1)

& Local variables:
& ----------------

INTEGER ibar, isubpart;
REAL sum;
STRING pref_datafile, ext_datafile, name_datafile;

BEGIN

  pref_datafile := "ph_";                        & data file name prefix
  ext_datafile  := ".dat";                       & data file name extension

  FILASSIGN (ph_gnuplotfile, filename);
  OPEN (ph_gnuplotfile, 3);

  WRITELN (ph_gnuplotfile, "set title '", title, "'");
  WRITELN (ph_gnuplotfile, "set boxwidth 0.5");
  WRITELN (ph_gnuplotfile, "set autoscale y");
  WRITELN (ph_gnuplotfile, "set xrange [0.5:", nb_bars + 0.5, "]");
  WRITELN (ph_gnuplotfile, "plot \");
  FOR ibar := 1 STEP 1 UNTIL nb_bars DO BEGIN

      name_datafile := pref_datafile //
                       CONVERT(ibar, STRING) // ext_datafile;
      FILASSIGN (ph_datafile, name_datafile);
      OPEN (ph_datafile, 3);

      sum := 0.;
      WRITELN (ph_datafile, ibar, " ", 0.0);

      FOR isubpart := 1 STEP 1 UNTIL nb_subparts DO BEGIN
          sum := sum + x (ibar, isubpart);
          WRITELN (ph_datafile, ibar, " ", sum);
      END;  & for

      CLOSE (ph_datafile);

      WRITE (ph_gnuplotfile, "'", name_datafile, "'");
      WRITE (ph_gnuplotfile, " title '", legend (ibar), "'");
      WRITE (ph_gnuplotfile, " with boxes");

      IF (ibar < nb_bars) THEN WRITE (ph_gnuplotfile, ", \");
      WRITELN (ph_gnuplotfile);

  END;  & for

  WRITELN (ph_gnuplotfile, "pause -1 'Press return to continue...'");
  CLOSE (ph_gnuplotfile);

END;  & proc
```

10.3 Example

Here is an example using the two preceding procedures with the corresponding GNUPLOT displays.

Example code:

```
/DECLARE/

INTEGER nb_curve = 4,
        nb_point = 10;

REAL x (nb_curve, nb_point), y (nb_curve, nb_point);

STRING legend (nb_curve);

INTEGER icurve, ipoint;

/EXEC/
BEGIN

   FOR icurve := 1 STEP 1 UNTIL nb_curve DO BEGIN
       FOR ipoint := 1 STEP 1 UNTIL nb_point DO BEGIN
           x (icurve, ipoint) := ipoint + RANDU;
           y (icurve, ipoint) := icurve + RANDU;
       END;
       legend (icurve) := "Stream " // CONVERT (icurve, STRING);
   END;

   GPLOCUR ("curve.gnuplot", nb_curve, nb_point,
            x, y, "Random curves", legend);
   PRINT ("Please run gnuplot on curve.gnuplot to display curves.");

END;  & exec

/DECLARE/

INTEGER nb_bar = 6,
        nb_subpart = 5;

REAL z (nb_bar, nb_subpart);

STRING legend2 (nb_bar);

INTEGER ibar, isubpart;

REAL barmax;

/EXEC/
BEGIN
```

```
   barmax:= nb_subpart * ( 1 + RANDU);
   FOR ibar := 1 STEP 1 UNTIL nb_bar DO BEGIN
       FOR isubpart := 1 STEP 1 UNTIL nb_subpart DO BEGIN
           z (ibar, isubpart) := barmax - (isubpart + RANDU);
       END;
       legend2 (ibar) := "Bar chart " // CONVERT (ibar, STRING);
   END;

   GPLOBAR ("bar.gnuplot", nb_bar, nb_subpart,
            z, "Random bar charts", legend2);
   PRINT ("Please run gnuplot on bar.gnuplot to display bar charts.");

END;  & exec

/END/
```

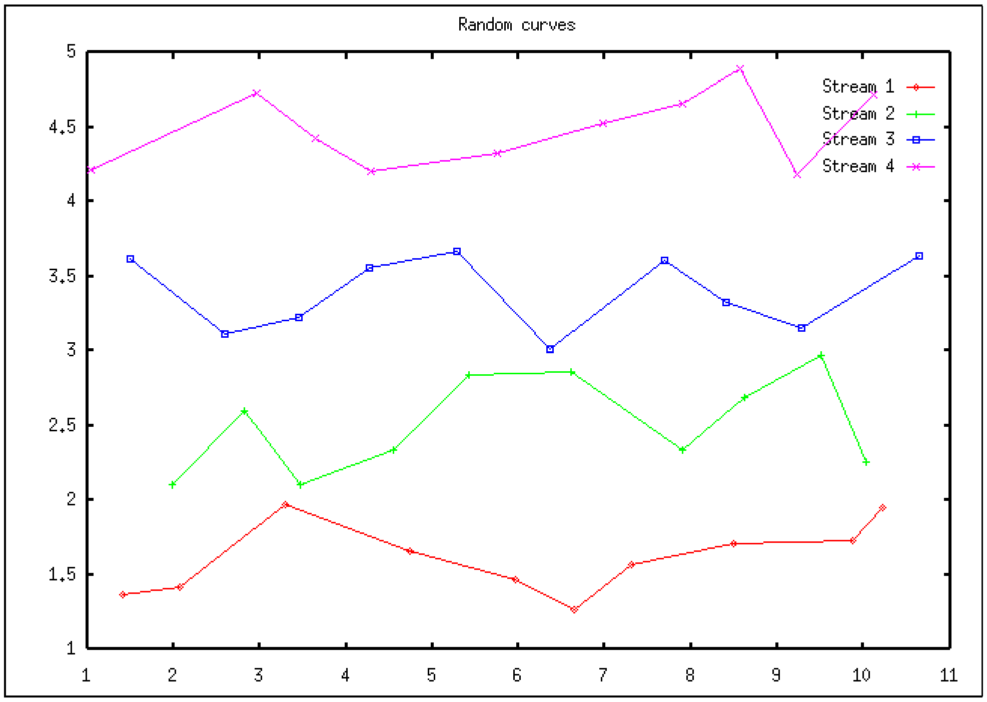


Figure 10.1: GNUPLOT curves plotting

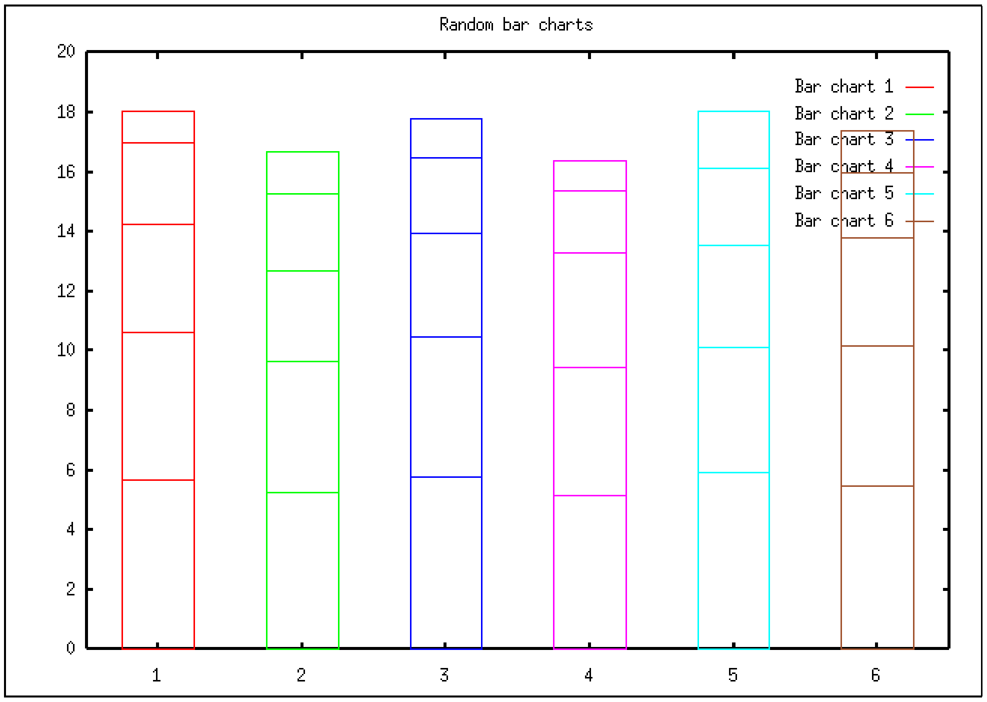


Figure 10.2: GNUPLOT bar charts plotting

□

Errors and Warnings $\boxed{\textbf{A}}$

A.1 Error management

A.1.1 Message format

The message printed when an error or a warning occurs is:

```
(0B0E05)  ==> ERROR (INTER) : END OF FILE ...
```

The first information displayed is an internal error code which helps locating the QNAP2 module issuing the message. These internal codes are used by QNAP2 developers to correct bugs.

The word `ERROR` or `WARNING` indicates the severity level of the reported problem. Errors cause the execution of the current /EXEC/ block to be abandonned. Warnings are just informational: execution continues after the message is printed.

The word between parentheses indicates which QNAP2 main module detected the error. For example, compilation errors are detected by the `COMPIL` module (the compiler), execution errors are most often detected by `INTER`, the algorithmic language interpreter. This is intended mostly for QNAP2 debugging, but can be useful to the user.

The rest of the message is the explicit error message. The message text is generally completed with additional information, such as the source line number and the values of the concerned variables.

Customers referenced in error messages are numbered according to their creation order. The numbering is performed modulo 1,000,000. Other objects are referred to by their identifier.

A.1.2 Qnap2 behaviour

A.1.2.1 Error during compilation

If a severe error occurs then the subsequent `/EXEC/` blocks are not processed and in particular, solvers are not called and saving is not possible. This is intended to avoid launching a long simulation or destroying data with an incorrect model.

A.1.2.2 Error during simulation

A severe error stops the simulation and aborts the /EXEC/ block containing the call to the SIMUL procedure.

A `SAVERUN` may be performed automatically if an error occurs during the simulation. The `SAVERUN` is performed if the user answers "Y" to the last question of the startup procedure:

```
DO YOU WANT QNAP2 TO MAKE A SAVE IN CASE OF ERROR (Y/N)? [N]
```

It is possible to restore the context of the simulation when the error occurred with the RESTORE procedure (see section 2.6.2).

The restored state is the state of the model just before the error occured. This facility is useful as a debugging tool. The cause of the error may be found in the `/REBOOT/` command.

Example:

```
/EXEC/ RESTORE ("QNAPSERR", "DEFERRED");

/REBOOT/ BEGIN
           FOR RQ:= ALL QUEUE DO
           PRINT (RQ, RQ.NB);
         END;
```

A.2 Error and Warning Messages

A.2.1 Introduction

This section describes the most important QNAP2 error messages. Each message is described in the following way:

- Message number,
- Message wording,
- Message explanation and action to be taken.

The messages are arranged following the lexical order of their numbers. **This is not an exhaustive list!** If the user ever meets an error message which is not clear enough, he/she is advised to contact SIMULOG (e-mail: *support@simulog.fr*) to get the information he/she needs.

Several message numbers can correspond to the same wording. It is due to the fact that an error message can be called at several places inside QNAP2 and that a message number correspond to one and only one of these calls.

A.2.2 List of messages

- 050101

  ```
  ==>ERROR (COMPILE) : INCORRECT SYNTAX
  ```

  The QNAP2 algorithmic language syntax is not respected. For example, IF without THEN or '(' without ')'...
  The user has to correct the line where the error occurred according to the algorithmic language syntax.

- 050103

  ```
  ==>WARNING (COMPILE) : ";" HAS BEEN ADDED BEFORE THIS ELEMENT
  ```

  QNAP2 adds ";" at the end of the preceding algorithmic statement. To avoid this warning message, the user has to add ";" in the source file.

- 050201

  ```
  ==>ERROR (COMPILE) : EXPRESSION IS NOT WELL DEFINED
  ```

  An expression in the algorithmic language is not correct. For example, an assignment operation may be wrong, because there is only one operand.
  The user has to check the syntax of the source lines.

- 050202

  ```
  == > ERROR (COMPILE) : INCORRECT MIXED TYPES
  ```

  This message means that a value of a specific type (integer, real, reference to an object) is assigned to a variable of another type. For example, when a WITH operation uses a reference the type of which is different from the type of the listed entities.
  The user has to ensure that the types of the concerned entities are the same, possibly by using the operators IS, IN or :: .

- 050203

  ```
  ==>ERROR (COMPILE) : INCORRECT TYPE FOR AN OPERAND OR AN ARGUMENT
  ```

  A data used in an operation or as a parameter of a procedure does not belong to the expected type.
  The user has to check the syntax of the performed operation and the types of the concerned entities.

- 050206

  ```
  ==>ERROR (COMPILE) : THIS IDENTIFIER HAS NOT BEEN DECLARED ...
                       OR IS NOT KNOWN IN THIS CONTEXT ...
  ```

An *identifier* is the name of a user-defined variable (integer, real, object type, user procedure, ...) or a QNAP2 key-word.
QNAP2 does not recognize the printed identifier in the model source code. The user has to declare the identifier, if it has not been already done, and/or to check its spelling.

- 050207

```
==>ERROR (COMPILE) : THIS VARIABLE CANNOT BE MODIFIED ...
```

It is forbidden to modify the internal variables used by QNAP2, such as the number of customers in a station or the address of an object entity.
The user has to check the validity of the variable modification.

- 050208

```
==>ERROR (COMPILE) : THIS IDENTIFIER CANNOT BE INDEXED ...
```

An indexed notation has got the form: `object.attribute`. The printed identifier corresponds to *object*. This message means that the *object* identifier has been defined as a simple variable, and not as an object instance.
The user has to modify the faulty statement or the *object* declaration.

- 05020F

```
==>ERROR (COMPILE) : "ALL" CANNOT BE USED FOR CUSTOMERS
```

The ALL key-word is used to build a list of entities of the same type. But CUSTOMER entities cannot be listed by this means, to prevent from building too large lists.
To build such a list, the user has to build a list of queues, and for each queue, access to its customers by: `queue.FIRST - customer.NEXT` ...

- 05020Q

```
==>ERROR (COMPILE) : THIS TYPE CANNOT QUALIFY THIS REFERENCE
```

The :: operator checks that a specific entity belongs to the specified type. This message means that the variable preceding the :: does not belong to the type specified after the :: .
The user has to check the model coherence.
Example:

```
/DECLARE/
  REF CUSTOMER @_c;
/EXEC/
BEGIN
  WITH @_c::QUEUE DO PRINT("???");
END;
/END/
```

- 05020U

```
==>ERROR (COMPILE) : CALL TO A GENERIC PROCEDURE FORBIDDEN
```

A generic procedure is different from the other ones because it has no algorithmic code, it only defines a procedure signature (the list of arguments and their types).
Such a procedure cannot be activated because there is no corresponding algorithmic code. It is necessary to use a reference on this procedure and to assign it to a regular procedure (the signature of which is equal to the one defined by the generic procedure).
The user has to see how to use generic procedures, see the key-words: `PROCEDURE - GENERIC - REF` .

- 050301

```
==>ERROR (COMPILE) : EXPRESSION IS NOT WELL DEFINED
```

An expression in the algorithmic language is not correct. For example, an assignment operation may be wrong, because there is only one operand.
The user has to check the syntax of the source lines.

- 050302

```
050302  ==> ERROR (COMPILE) : INCORRECT MIXED TYPES
```

This message means that a value of a specific type (integer, real, reference to an object) is assigned to a variable of another type. For example, when a `WITH` operation uses a reference the type of which is different from the type of the listed entities.
The user has to ensure that the types of the concerned entities are the same, possibly by using the operators `IS`, `IN` or `::` .

- 050303

```
==>ERROR (COMPILE) : INCORRECT TYPE FOR AN OPERAND OR AN ARGUMENT
```

A data used in an operation or as a parameter of a procedure does not belong to the expected type.
The user has to check the syntax of the performed operation and the types of the concerned entities.

- 050304

```
==>ERROR (COMPILE) : INCORRECT NUMBER OF ARGUMENTS
```

The arguments of a procedure are not valid. The error may be due either to the types of the arguments or to their number.
The user has to check the syntax of the concerned procedure and to modify the model according to it.

- `050403`

  ```
  ==>ERROR (COMPILE) : INCORRECT TYPE FOR AN OPERAND OR AN ARGUMENT
  ```

  A data used in an operation or as a parameter of a procedure does not belong to the expected type.
  The user has to check the syntax of the performed operation and the types of the concerned entities.

- `050404`

  ```
  ==>ERROR (COMPILE) : INCORRECT NUMBER OF ARGUMENTS
  ```

  The arguments of a procedure are not available. The error may be due either to the types of the arguments or to their number.
  The user has to check the syntax of the concerned procedure and to modify the model according to it.

- `060101`

  ```
  ==>ERROR (CONTROL) : INCORRECT SYNTAX
  ```

  The `QNAP2` language syntax is not respected in a `/CONTROL/` block.
  The `QNAP2` algorithmic language syntax is not respected. For example, `IF` without `THEN` or '(' without ')'...
  The user has to correct the line where the error occurred according to the algorithmic language syntax.

- `060103`

  ```
  ==>ERROR (CONTROL) : CLASS MAX. NUMBER CAN NO LONGER BE EXTENDED
  ```

  The default maximum number of customer classes is 20. This value can be modified by the command: `/CONTROL/ NMAX= ...`. The error message is printed if queues and/or classes have already been defined when the user tries to modify this value.
  The user is allowed to modify the maximum number of classes only if neither queues nor classes have been defined before.

- `060105`

  ```
  ==>WARNING (CONTROL) : ";" HAS BEEN ADDED BEFORE THIS ELEMENT
  ```

  `QNAP2` adds ";" at the end of the preceding algorithmic statement. To avoid this warning message, the user has to add ";" in the source file.

- `060107`

  ```
  ==>ERROR (CONTROL) : BAD FILE OR LOGICAL UNIT SPECIFICATION
  ```

  `/CONTROL/ UNIT =` *type(file)* command is used to modify the output file. This message means that the new output file must be explicitly defined.

The user has to modify the command or (better) to use procedures (e.g., FILASSIGN) to manage files.

- 060108

```
==>ERROR (CONTROL) : ATTEMPT TO ASSIGN OUTPUT TO A FILE CLOSED
                     OR OPEN IN A WRONG MODE. IGNORED
```

/CONTROL/ UNIT = *type(file)* command is used to modify the input-output files. This message means that the new output file (FSYSOUTP) is closed or opened in “read” mode.
The user has to modify the command or (better) to use procedures (FILASSIGN - OPEN) to manage files.

- 06010C

```
==>ERROR (CONTROL) : ATTEMPT TO ASSIGN INPUT TO A FILE CLOSED
                     OR OPEN IN A WRONG MODE. IGNORED
```

/CONTROL/ UNIT = *type(file)* command is used to modify the input-output files. This message means that the new input file (FSYSINPU, where the modele code is read) is closed or opened in “write” mode.
The user has to modify the command or (better) to use procedures (FILASSIGN - OPEN) to manage files.

- 06010D

```
==>ERROR (CONTROL) : ATTEMPT TO ASSIGN PRINT TO A FILE CLOSED
                     OR OPEN IN A WRONG MODE. IGNORED
```

/CONTROL/ UNIT = *type(file)* command is used to modify the input-output files. This message means that the new file specified to write the user messages (FSYSPRINT) is closed or opened in “read” mode.
The user has to modify the command or (better) to use procedures (FILASSIGN - OPEN) to manage files.

- 06010E

```
==>ERROR (CONTROL) : ATTEMPT TO ASSIGN GET TO A FILE CLOSED
                     OR OPEN IN A WRONG MODE. IGNORED
```

/CONTROL/ UNIT = *type(file)* command is used to modify the input-output files. This message means that the new file specified to read data (FSYSGET) is closed or opened in “write” mode.
The user has to modify the command or (better) to use procedures (FILASSIGN - OPEN) to manage files.

- 060201

```
==>ERROR (DECLARE) : THIS IDENTIFIER HAS ALREADY
                     BEEN DECLARED ...
```

An *identifier* is the name of a user-defined variable (integer, real, object type, user procedure, ...) or a QNAP2 key-word.
This message is printed when the user tries to declare an identifier that has already been defined. The user has to check the previous declarations and the identifier spelling.

- 060202

  ```
  ==>ERROR (DECLARE) : INCORRECT STRING LENGTH  ...
  ```

  A STRING variable is defined by:
  `STRING [ (length) ] id [= string ] ;`
  where `length` is an integer representing the maximum number of characters in the string.
  This message is printed if the number of characters declared for the string is less than 0 or greater than 256.
  The user has to check the syntax of strings declaration and the integer value given for their length.

- 060302

  ```
  ==>ERROR (DECLARE)  : INCORRECT SYNTAX
  ```

  The QNAP2 algorithmic language syntax is not respected. For example, IF without THEN or '(' without ')'...
  The user has to correct the line where the error occurred according to the algorithmic language syntax.

- 060303

  ```
  ==>ERROR (DECLARE) : THIS TYPE IS UNKNOWN ...
  ```

  To declare an object instance, the object type identifier is specified before the name of the instance. If an unknown identifier is detected at the begining of a declaration sentence, the current message is printed.
  The user has to check the concerned identifier syntax.
  Example:

  ```
        1 /DECLARE/ foo bar;
                      |
   (060303)  ==>ERROR (DECLARE) : THIS TYPE IS UNKNOWN ... foo
        2
        3 /END/
  STOP: QNAP2 : END OF EXECUTION
  ```

- 060306

  ```
  ==>ERROR (DECLARE) : THIS VARIABLE CANNOT BE INITIALIZED
  ```

  This error appears in a DECLARE block when a variable, declared as an object attribute, or a protected variable, is initialized.

The user has to modify the declaration.
Example:

```
      1 /DECLARE/ OBJECT foo;
      2             INTEGER I=4;
                             |
 (060306)  ==>ERROR (DECLARE) : THIS VARIABLE CANNOT BE INITIALIZED
      3           END;
                    |
 (060302)  ==>ERROR (DECLARE) : INCORRECT SYNTAX
      4
      5 /END/
STOP: QNAP2 : END OF EXECUTION
```

- 060307

```
 ==>ERROR (DECLARE) : ATTRIBUTES CANNOT BE DECLARED
                      IF OBJECTS OF THE CORRESPONDING TYPE
                      HAVE BEEN CREATED
```

Attributes cannot be added to an object type if objects of this type already exist. The user has to modify the model structure or to create two different types.

- 060309

```
 ==>WARNING (DECLARE) : ";" HAS BEEN ADDED BEFORE THIS ELEMENT
```

QNAP2 adds ";" at the end of the preceding algorithmic statement. To avoid this warning message, the user has to add ";" in the source file.

- 06030J

```
 ==>ERROR (DECLARE) : "ANY" IS NOT A LEGAL IDENTIFIER
```

The ANY keyword can be used only after the REF keyword to define a reference to any object type. This message is printed if ANY is used without the REF keyword. The user has to modify the declaration.
Example:

```
      1 /DECLARE/ QUEUE ANY @_ptr;
                          |
 (06030J)  ==>ERROR (DECLARE) : "ANY" IS NOT A LEGAL IDENTIFIER
      2
      3 /END/
```

- 06030L

```
 ==>ERROR (DECLARE) : THIS IDENTIFIER HAS ALREADY
                      BEEN DECLARED ...
```

An *identifier* is the name of a user-defined variable (integer, real, object type, user procedure, ...) or a QNAP2 key-word.

This message is printed when the user tries to declare an identifier that has already been defined. The user has to check the previous declarations and the identifier spelling.

- 060601

```
==>ERROR (STATION) :  INCORRECT SYNTAX
```

The `QNAP2` language syntax is not respected in a `/STATION/` block.
The `QNAP2` algorithmic language syntax is not respected. For example, `IF` without `THEN` or '(' without ')'...
The user has to correct the line where the error occurred according to the algorithmic language syntax.

- 060606

```
==>WARNING (STATION) : ";" HAS BEEN ADDED BEFORE THIS ELEMENT
```

`QNAP2` adds ";" at the end of the preceding algorithmic statement. To avoid this warning message, the user has to add ";" in the source file.

- 0A0101

```
==>ERROR (EDIT) : INCORRECT SYNTAX
```

The `QNAP2` algorithmic language syntax is not respected. For example, `IF` without `THEN` or '(' without ')'...
The user has to correct the line where the error occurred according to the algorithmic language syntax.

- 0A0102

```
==>WARNING (EDIT) : END OF FILE DETECTED ON FILE ...
```

This error is generated when the input file has been entirely read and no end of model has been specified.
It is usually due to the fact that the `/END/` key-word has been forgotten at the end of the source file. The user only has to add `/END/` at the end of the file.

- 0A0105

```
==>ERROR (EDIT) : MEMORY OVERFLOW
```

This message means that the input file cannot be read because `QNAP2` memory space is full.
The user can either try to reduce the size of its model (too many queues or too many customers) or use a `QNAP2` executable with a larger memory space. To get such a `QNAP2` executable, the user can generate it himself or ask SIMULOG for it. In both cases, it is advisable to contact SIMULOG to get more information.

- 0A0201

```
 ==>ERROR (EDIT) : INCORRECT SYNTAX
```

The `QNAP2` algorithmic language syntax is not respected. For example, `IF` without `THEN` or '(' without ')'...
The user has to correct the line where the error occurred according to the algorithmic language syntax.

- 0A0205

```
==>ERROR (EDIT) : MEMORY OVERFLOW
```

This message means that the input file cannot be read because `QNAP2` memory space is full.
The user can either try to reduce the size of its model (too many queues or too many customers) or use a `QNAP2` executable with a larger memory space. To get such a `QNAP2` executable, the user can generate it himself or ask SIMULOG for it. In both cases, it is advisable to contact SIMULOG to get more information.

- 0A0501

```
==>ERROR (COMPILE) : THIS IDENTIFIER HAS NOT BEEN
                     DECLARED ...
```

An *identifier* is the name of a user-defined variable (integer, real, object type, user procedure, ...) or a `QNAP2` key-word.
`QNAP2` does not recognize the printed identifier in the model source code. The user has to declare the identifier, if it has not been already done, and/or to check its spelling.

- 0B0902

```
==>ERROR (INTER) : FILE ...  IS NOT WELL BUILT
```

This message is printed when an input/output operation is performed on a file which has not been assigned.
The user has to assign and open (`FILASSIGN` and `OPEN` procedures) the file.
Example:

```
      1 /DECLARE/ FILE f1;
      2           INTEGER I;
      3 /EXEC/ I:=GET (f1,INTEGER);
 (0B0C04)  ==>ERROR (INTER) : FILE ... f1       IS NOT WELL BUILT
 (0I0500)                     LINE NUMBER :      3
      4 /END/
STOP: QNAP2 : END OF EXECUTION
```

- 0B0905

```
==>ERROR (INTER) : CANNOT ASSIGN OPEN FILE ...
```

The procedure `FILASSIGN` assigns a physical file (the name of which is given as second argument) to the file object given as first argument.

The preceding error message is printed if the QNAP2 file is already opened. The user has to check that the specified file is closed before assigning it.

- 0B0A02

```
==>WARNING (INTER) : SYSTEM FILE ...  WILL NOT BE CLOSED
```

The QNAP2 procedure CLOSE is used to close a file. Be careful that the predeclared file FSYSINPU and the implicit files cannot be closed.
To suppress this warning message, the user has to check that he/she is not trying to close an implicit file.

- 0B0A05

```
==>ERROR (INTER) : BAD FILE DEFINITION IN CLOSE PROCEDURE
```

This message is printed if a QNAP2 file is closed without having been assigned before.
The user has to assign the file or to ensure that the file has been opened.
Example:

```
      1 /DECLARE/ FILE f1;
      2 /EXEC/ CLOSE (f1);
 (0B0A05)  ==>ERROR (INTER) : BAD FILE DEFINITION IN CLOSE PROCEDURE
 (0I0500)                     LINE NUMBER :      2
      3 /END/
STOP: QNAP2 : END OF EXECUTION
```

- 0B0A07

```
==>ERROR (INTER) : UNABLE TO CLOSE FILE ...
```

This message is printed if QNAP2 cannot close a file using the CLOSE procedure.
The reason is given in a following message.
The problem is often due to bad access rights to the considered file for the user.

- 0B0C01

```
==>ERROR (INTER): THE DATA WHICH HAS BEEN GIVEN IS NOT CORRECT
                  TRY AGAIN ...
                  ... LINE NUMBER :
```

The QNAP2 procedures GET and GETLN read values of specific types on a file.
The current message means that the data being read do not belong to the expected type. New read operations can be performed depending on the value of the ERRRETRY file attribute.
The user is advised to check that the read data belong to the expected type.

- 0B0C04

```
==>ERROR (INTER) : FILE ...  IS NOT WELL BUILT
```

This message is printed when an input/output operation is performed on a file which has not been assigned.
The user has to assign and open (`FILASSIGN` and `OPEN` procedures) the file.

- `0B0C08`

```
==>ERROR (INTER) : FORMAT WIDTH EXCEEDS BUFFER SIZE.
                   FILE ...
```

The `QNAP2` procedures `GET` and `GETLN` read a value on a file with a specific format. The format specifies the number of characters to be read. This number must be lower than the buffer size.
The user has to modify the reading format in order to clear this error message.

- `0B0E05`

```
==>ERROR (INTER) : END OF FILE DETECTED ON FILE ...
```

This message means that a reading operation is requested on a file that has been entirely read.
The user has to check the file contents and the reading operations already performed on it.

- `0B0F07`

```
==>ERROR (INTER) : UNABLE TO OPEN FILE ...
```

This message is printed when `QNAP2` does not succeed in opening a file because of the system.
The user has to check he can access the file.

- `0B0G05`

```
==>ERROR (SUPER) : UNABLE TO OPEN FILE FOR RESTORE OPERATION
```

This message is printed when `QNAP2` does not succeed in opening a file in which an execution context has been saved.
The user has to check he can access the file.

- `0B0G0A`

```
==>ERROR (SUPER) : MODEL NOT FOUND ON RESTORE FILE ...
```

This message means that the restore operation failed for one of the following reasons:

- the string specified for the save operation is different from the string specified for the restore operation,
- an error occurred during the file reading.

The user has to:

- read the file attribute `ERRSTATUS` to get more information about the error,

- check that the strings specified for the SAVE and RESTORE operations are the same.

- 0B0G0B

```
==>ERROR (SUPER) : SPACE SIZES NOT IDENTICAL :
                            WORDS FOR "SAVE",
                            WORDS "RESTORE"
```

The QNAP2 versions used for the model save and restore are different, the difference is the size of the QNAP2 memory space.
The user has to modify the size of the QNAP2 memory space, so that the QNAP2 executable used to restore the model has the same size as the executable used to save it.

- 0B0G0D

```
==>ERROR (SUPER) : VERSION OF QNAP2 HAD CHANGED BETWEEN
                   SAVE AND RESTORE. YOU MUST SAVE
                   YOUR MODEL AGAIN WITH THE NEW VERSION
```

The QNAP2 versions used for the model save and restore are different.
It is mandatory that the save and restore operation be performed by the same QNAP2 version.

- 0B0H07

```
==>ERROR (SUPER) : ABORT: THE NETWORK IS PROBABLY SATURATED
```

This message occurs when the execution context cannot be saved because the QNAP2 memory space is full.
This message means that the input file cannot be read because QNAP2 memory space is full.
The user can either try to reduce the size of its model (too many queues or too many customers) or use a QNAP2 executable with a larger memory space. To get such a QNAP2 executable, the user can generate it himself or ask SIMULOG for it. In both cases, it is advisable to contact SIMULOG to get more information.

- 0B0I03

```
==>ERROR (INTER) : FILE ...  IS NOT WELL BUILT
```

This message is printed when an input/output operation is performed on a file which has not been assigned.
The user has to assign and open (FILASSIGN and OPEN procedures) the file.

- 0G0102

```
==>ERROR (EVAL) : NO STATION IN THE NETWORK
```

A network checking occurs when a resolution method (`MARKOV - SIMUL- SOLVE`) is invoked. But a resolution has no meaning if no station is defined.
The user has to check that queues are created before the model resolution.

- 0G0108

```
 ==>ERROR (EVAL) : TRANSITION TO AN UNDEFINED QUEUE
```

This message means that the customers leaving a queue are sent to an undefined station. This message is printed only if the transition is defined with a reference to a station equal to NIL.
Example:

```
      1 /DECLARE/ QUEUE OBJECT foo;
      2            INTEGER N;
      3           END;
      4           REF foo D;
      5           QUEUE A,B,C;
      6           REAL r1=-1;
      7
      8 /STATION/ NAME=A;
      9           INIT=1;
     10           SERVICE= EXP(1);
     11           TRANSIT=B,0.5,C,r1,D;
 [...]
     23 /EXEC/ BEGIN
     24          r1:=0.1;
     25          SOLVE;
     26        END;
 (0G0108)  ==>ERROR (EVAL) : TRANSITION TO AN UNDEFINED QUEUE
                              ... STATION : A
     27
     28 /END/
STOP: QNAP2 : END OF EXECUTION
```

- 0G0109

```
 ==>ERROR (EVAL) : TRANSITION TO A QUEUE NOT IN THE NETWORK
```

The `NETWORK` procedure is used to specify a sub-network, in this case, the resolution will apply only to the stations in the sub-network.
This message means that some customers are sent to a station which does not belong to this sub-network.
The user can remove the `NETWORK` procedure from the model, extend the list of queues in the sub-network or modify the customer transitions so that they stay in the sub-network.

- 0G010B

```
 ==>ERROR (EVAL) : TRANSITION TO AN UNDEFINED CLASS
```

This message means that the class of customers leaving a queue is changed to an undefined class.
This message is printed only if the class is defined with a reference equal to NIL.
Example:

```
      1 /DECLARE/
      2          REF CLASS @_C;
      3          CLASS C1,C2;
      4          QUEUE A,B,C;
      5          REAL r1=-1;
      6
      7 /STATION/ NAME=A;
      8          INIT=1;
      9          SERVICE= EXP(1);
     10          TRANSIT=B,0.5,C,@_C,r1;
 [...]
     22 /EXEC/ BEGIN
     23         r1:=0.1;
     24         SOLVE;
     25       END;
 (0G010B)  ==>ERROR (EVAL) : TRANSITION TO AN UNDEFINED CLASS
                               ... STATION : A       CLASS :
     26
     27 /END/
```

- 0G010K

```
 ==>ERROR (EVAL) : A SOURCE, A RESOURCE OR A SEMAPHORE
                   CANNOT BE INITIALIZED WITH CUSTOMERS
```

The `INIT` parameter, in a `/STATION/` block, specifies the number of customers in a queue before the beginning of the resolution.
Source, resource or semaphore stations take specific roles and cannot contain customers before the resolution begins. Such stations are defined by the following commands:

- `TYPE = SOURCE;`
- `TYPE = RESOURCE;`
- `TYPE = SEMAPHORE;`

The user has to suppress the `INIT` command or modify the station type.

- 0I0507

```
 ==>ERROR (INTER) : INDEX OUT OF BOUNDS. VALUE :
```

This message means that the index computed by `QNAP2` is out of the array bounds. The user has to compare the index value with the bounds declared for the array.

- 0I050B

```
 ==>WARNING (INTER) : ATTEMPT TO DIVIDE BY ZERO,
                   MAXIMUM VALUE ASSUMED
```

A division by zero has been detected by QNAP2 in the model algorithmic code. The result of the operation is then equal to the maximum real or integer value the machine can manage.
It is highly advisable not to allow such operations.

- 0I050H

```
==>ERROR (INTER) : INCORRECT CONTROL VALUES IN A "STEP/UNTIL"
                   CLAUSE, THE SECOND BOUND CANNOT BE REACHED
```

This message is printed if the higher bound of a STEP UNTIL command is lower than the lower bound.
It is mandatory to modify the values of the bounds.

- 0I051D

```
==>ERROR (INTER) : OBJECT REFERENCED HERE DOES NOT BELONG
                   TO THE TYPE ...  EXPECTED FOR THIS
                   REFERENCE OR IS ALREADY DELETED ...
```

This message means that a reference refers to an object which does not belong to the specified type, or which has been deleted.
The user has to check the reference type using the IS or IN operators.
Example:

```
      1 /DECLARE/ QUEUE A,B;
      2           CUSTOMER OBJECT person;
      3             REAL d_entry;
      4           END;
      5           REF person @person;
      6           REF ANY rc;
      7
[...]
     17 /STATION/NAME=A;
     18          TRANSIT=OUT;
     19          SERVICE=BEGIN
     20                    CST(2);
     21                    rc:=QUEUE;
     22                    WITH rc::person DO
     23                      PRINT("d_entry: ",d_entry);
     24                   END;
[...]
*** SIMULATION ***
(0I0508)  ==>ERROR (INTER) : OBJECT REFERENCED HERE DOES NOT BELONG
                             TO THE TYPE ... person   EXPECTED FOR THIS
                             REFERENCE OR IS ALREADY DELETED ...
(0I0500)                     LINE NUMBER :     22
... ACTIVE STATION : A        FOR CUSTOMER :      2(        )
... TIME =      2.000
```

```
                    LINE NUMBER :      22
     30
     31 /END/
STOP: QNAP2 : END OF EXECUTION
```

- 0I0603

```
==>WARNING (INTER) : RESULT NOT AVAILABLE ... ZERO ASSUMED
```

This message is printed if a non computed statistical result is requested by the user. The result is not computed because the sample is empty or because its computation has not been requested.
The user has to ask for the result computation using the appropriate `SETSTAT:` procedure and not to request non computed results.

- 0I0701

```
==>ERROR (INTER) : INVALID VALUE SPECIFIED FOR TMAX
```

This message is printed if the `SETTMAX` procedure is used with an invalid parameter: lower than zero or lower than the current date.
The user has to check the value of the parameter of the `SETTMAX` procedure.

- 0I0702

```
==>ERROR (INTER) : MEMORY OVERFLOW
```

This message means that the input file cannot be read because `QNAP2` memory space is full.
The user can either try to reduce the size of its model (too many queues or too many customers) or use a `QNAP2` executable with a larger memory space. To get such a `QNAP2` executable, the user can generate it himself or ask SIMULOG for it. In both cases, it is advisable to contact SIMULOG to get more information.

- 0I0B01

```
==>ERROR (INTER) : A REFERENCE WITH VALUE "NIL" IS USED
```

This message is printed if we use a reference that refers to an object equal to NIL.
The user has to modify the reference value, so that it refers to an existing object.

- 0I0B02

```
==>ERROR (INTER) : A REFERENCE TO A DESTROYED OBJECT IS USED
```

This message means that a reference to a deleted object is used.
The user has to modify the reference value, so that it refers to an existing object.
He/she can check the referenced object destruction using the `DELETED` function.

- 0J0501

```
==>ERROR (SUPER) : MEMORY OVERFLOW ...
```

This message means that the input file cannot be read because `QNAP2` memory space is full.
The user can either try to reduce the size of its model (too many queues or too many customers) or use a `QNAP2` executable with a larger memory space. To get such a `QNAP2` executable, the user can generate it himself or ask SIMULOG for it. In both cases, it is advisable to contact SIMULOG to get more information.

- `0J0602`

```
==>ERROR (FREELM) : DOUBLE FREE ON THE SAME AREA
                    BUG IN QNAP2
```

The same `QNAP2` memory area has been freed twice. It is a `QNAP2` internal error. The user has to contact SIMULOG (e-mail: *support@simulog.fr*, tel: +33-(1)-30-12-27-77).

- `0J0901`

```
==>ERROR (SUPER) : MEMORY OVERFLOW
                   ... (GETLIM)
                   ... ALREADY ALLOCATED :  WORDS
                   ... STILL AVAILABLE   :  WORDS
                   ... REQUESTED         :  WORDS
```

This message means that the input file cannot be read because `QNAP2` memory space is full.
The user can either try to reduce the size of its model (too many queues or too many customers) or use a `QNAP2` executable with a larger memory space. To get such a `QNAP2` executable, the user can generate it himself or ask SIMULOG for it. In both cases, it is advisable to contact SIMULOG to get more information.

- `0J0H02`

```
==>ERROR (SUPER) : INCORRECT SYNTAX
```

The `QNAP2` algorithmic language syntax is not respected. For example, `IF` without `THEN` or '(' without ')'...
The user has to correct the line where the error occurred according to the algorithmic language syntax.

- `0J0H08`

```
==>WARNING (SUPER) : ";" HAS BEEN ADDED BEFORE THIS ELEMENT
```

`QNAP2` adds ";" at the end of the preceding algorithmic statement. To avoid this warning message, the user has to add ";" in the source file.

- `0J0H09`

```
==>WARNING (SUPER) : SOME STATEMENTS ARE SKIPPED
                     UNTIL SOME COMMAND OCCURS
```

This message is printed after every message that occurs during the analysis of a command content. This message warns the user that the analysis of the model goes to the next command block.
The user has to correct the error to get a complete analysis of the model.

- 0J0H0A

```
==>ERROR (SUPER) : CANNOT LAUNCH EXECUTION
                   SOME FATAL ERRORS OCCURED
```

This message is printed if the compilation of the algorithmic code of the `/EXEC/` command has failed.
The user has to correct the compilation errors.

- 0R0101

```
==>WARNING (SIMUL) : ARITHMETIC UNDERFLOW ON TIME MANAGEMENT
                     A SERVICE DELAY IS TOO SMALL
                     COMPARED WITH "TIME"
```

The `QNAP2` real variables are in simple precision. Internally, the simulation time is managed in double precision. So, if you have a long simulation time with small delays, there may be losses of precision on the user-managed variables.
The preceding message means that the user will have to take care of the validity of the manually computed time-dependent results.
The user is advised to use the standard results computed by `QNAP2`, and not to build simulation models with two different time scales (because of the validity of the user-computed results and to avoid too long simulations).

- 0R0408

```
==>ERROR (SIMUL) : NEGATIVE VALUE ASSIGNED
                   TO A DELAY :
```

A `CUSTOMER` or a `TIMER` is asked to wait a negative time delay, which is forbidden.
The user has to ensure that this delay be always positive.

- 0R0409

```
==>ERROR (SIMUL) : UNDEFINED TRANSITION
```

No transition has been defined for the `CUSTOMER` in the specified `QUEUE`.
The user has to check the `TRANSIT` parameter of the queue.

- 0R0A01

```
==>WARNING (SIMUL) : NO TMAX SPECIFIED
                     TMAX=0. ASSUMED
```

The TMAX parameter has not been specified by the user at the beginning of the simulation. QNAP2 will assume that TMAX is null.
The user has either to change the TMAX value at the simulation beginning (with the SETTMAX procedure) or to add a /CONTROL/ TMAX=...; instruction before the simulation launching.

- 0R0M03

```
==>ERROR (SIMUL) : MEMORY OVERFLOW
```

This message means that the input file cannot be read because QNAP2 memory space is full.
The user can either try to reduce the size of its model (too many queues or too many customers) or use a QNAP2 executable with a larger memory space. To get such a QNAP2 executable, the user can generate it himself or ask SIMULOG for it. In both cases, it is advisable to contact SIMULOG to get more information.

- 130L02

```
==>ERROR (STATIS) : NO STATISTIC STRUCTURE CONNECTED
                    TO THE VARIABLE
```

The user tries to obtain a result (using a GETSTAT function) on a variable to which no statistic structure is connected.
The user has to check that the GETSTAT function arguments are queues or watched variables for which the specified result has been requested.

- 130L03

```
==>ERROR (STATIS) : UNAVAILABLE RESULT BECAUSE MISSING USER
                    SPECIFICATION
```

The user tries to obtain a result (using a GETSTAT function) the calculation of which has not been requested.
The user has to check that the GETSTAT function arguments are queues or watched variables for which the specified result has been requested.

- 130M0C

```
==>ERROR (STATIS) : NO STATISTIC ON THE STATION
```

The user tries to obtain a queue result (using a GETSTAT function) the calculation of which has not been requested.
The user has to check that the SETSTAT request corresponding to this result has been performed.

- 130M0J

```
==>ERROR (STATIS) : OPERATION FORBIDDEN DURING SIMULATION
```

Most of the SETSTAT procedures can not be called during the simulation.
The user has to perform the results requests before the simulation starts.

A.3 Problem reporting

Sometimes, QNAP2 does not behave as expected. Most often, a careful analysis of the model is required to understand what happened in the model.

You may also run into a bug. Generally, you will find that a slight modification in your model makes the problem disappear and allows you to go on. However, we would like to hear about all problems you encounter with QNAP2, in order to fix bugs and make the product better.

We would also be pleased to hear about problems with the documentation.

When you think there is a bug, and you want to report it, make sure you provide us with all the necessary information. The very best information is a short model reproducing the problem, with a listing of the output produced by QNAP2. If this is not possible, then it will be very difficult for us to understand what happened and correct the problem.

When you report a bug, you should also fill in the bug report form on next page. This basic information will help us register the bug and track it.

QNAP2 PROBLEM REPORT	
Status:	Ref:
Date:	QNAP2 version:
Name:	Hardware:
Organization:	Operating system:
Phone:	E-mail:
Fax:	
Address:	
Suspected problem origin: (tick one)	
$\boxed{1}$ Software $\boxed{2}$ Documentation	$\boxed{3}$ Environment
Symptoms:	
$\boxed{1}$ Abort	$\boxed{4}$ Unexpected results
$\boxed{2}$ QNAP2 error message	$\boxed{5}$ Other:
$\boxed{3}$ System error message	
Severity level: (tick one)	
$\boxed{1}$ Blocking	$\boxed{3}$ Minor
$\boxed{2}$ Major (e.g., unavailable functionality)	$\boxed{4}$ Nice to have (e.g., spelling error)

Description: (please include a sample model)

New features $\boxed{\textbf{B}}$

This appendix presents the new features of QNAP2 release 9.0. Experienced users should read this appendix in order to survey the differences between this version and V8.0.

The main objective of the new features is to reduce the effort required to develop a model of a complex system.

Some new features (e.g., finite capacity queues) could be emulated with previous QNAP2 releases, but this was at the expense of programming efforts with the algorithmic language (typically: statements embedded in the service descriptions). Other features could not be emulated, even with complex algorithmic code (e.g., user functions), or would have required very heavy algorithmic code (e.g., concurrency sets).

1. New modelling mechanisms:
   - Finite capacity queues.
   - Multiple server stations with concurrency sets of customers.
   - Customer resequencing.
   - Customer split and match mechanisms.
   - Timers.
2. Simulation control mechanisms:
   - Generalized trace
   - Generalized statistics
   - Accuracy control
   - Exceptions
3. Extended analytical solvers, handling:
   - Finite capacity queues.
   - Multiple server stations with concurrency sets of customers.
4. Algorithmic language extensions:
   - User functions.
   - Hierarchical predeclared procedures and functions.

The new features are presented below. A few known compatibility problems with V8.0 are listed at the end of this chapter.

B.1 New modelling mechanisms

The new modelling mechanisms are intended to make modelling simpler to QNAP2 users.

B.1.1 Finite capacity queues

Most real world devices have a limited capacity. For example:

- A node buffer in a telecommunications network can hold a limited number of packets.
- A conveyor in a manufacturing plant can hold a limited number of items.

In V9.0, it is now possible to specify the maximum number of customers that can reside simultaneously in a queue. The default queue capacity is infinite. The maximum capacity can be specified globally and/or for each customer class.

When a customer attempts to enter a full queue, it is said to be *rejected.* The rejection mechanism is entirely user-controlled. It is defined by an algorithmic code sequence, very similar to the service process of the station.

The finite capacity queues are descibed in section 4.2.3, “Queue capacity”.

B.1.2 Concurrency sets

In a multiple-server station, several customers may be served simultaneously. Sometimes, it is necessary to prevent this, for example:

- When the customers represent parallel processes with precedence constraint.
- When two customers represent two processes sharing the same resource.

The general problem is that of *mutual exclusion.*

QNAP2 can model mutual exclusion with the concept of *concurrency sets.* With regard to each station, customers can be classified into several sets. The sets can be defined with customer classes and/or probabilities. Two customers belonging to the same concurrency set cannot be served simultaneously, even if there is an idle server.

This mechanism is implemented as a new scheduling option. It is compatible with the existing ones (i.e., FIFO, LIFO, PRIOR, etc.)

The concurrency sets are described in section 4.5.4, “Multiple server station with concurrency sets”.

B.1.3 Customer resequencing

It is often required to process customers in a specific order. For example:

- In a telecommunications network, packets arriving at their destination node must be resequenced before reassembling the message.
- In a distributed data processing system, data consistency requires that sequences of operations be performed in the same order on all nodes.

Resequencing is a new scheduling option. Customers arriving at the resequencing queue are served in the same order as they departed from a reference queue. Customers arriving out of sequence must wait until all their preceding customers have arrived.

This feature is described in section 4.5.5, “Resequencing station”.

B.1.4 Customer split and match

These new features are intended to model systems including parallel processing with *rendez-vous* mechanisms, for example:

- Message packetization and reassembly in telecommunication networks.

- *Fork* and *join* primitives in multi-tasking systems.
- Inter-process synchronizations.
- Assembly and disassembly of parts in a manufacturing process.

These mechanisms could be emulated with the algorithmic language with customer creations and synchronization procedures (e.g., NEW and JOIN). Four new mechanisms have been introduced in order to simplify this. The four keywords below are implemented as new parameters of the /STATION/ command.

SPLIT and MATCH should be used together. SPLIT breaks down a customers into pieces. MATCH joins all or part of the pieces together.

FISSION breaks a customer into pieces which cannot be joined up. It is similar to SPLIT, but the information about the original single customer is not kept.

FUSION joins any set of customers into one.

The split/match features are described in section 4.10, "Customer split and match".

B.1.5 Timers

Timers are new simulation processes. Simple timers could be emulated in previous versions of QNAP2 with queues and the algorithmic language. In V9.0, timers provide a simple and efficient way to control time-related events independently from the queueing network.

Timers can be used, for example, to:

- control response times,
- schedule periodic events,
- schedule rare events at a controlled date (e.g., a device failure).

Timers can be activated at a given absolute or relative time, or periodically. A procedure called the *handler* is associated to each timer. When a timer reaches its activation time, the handler is automatically called by QNAP2. The handler is a user written procedure, enabling full control of the model with all the power of the algorithmic language.

B.2 Simulation control

B.2.1 Generalized trace

The treatment of the events has been generalized. It is no more limited to a standard trace message. Any user function can be invoked on a simulation event. The function is executed just after the event has occured.

To define a trace function, a user must be able to access all the information edited by the standard trace. A set of new standard functions has been created to provide a reference on the entities involved in the simulation event, such as: the concerned QUEUE, the current FLAG or the number of customers involved in the event. The standard message can be mimicked by using these functions.

A user may be interested only in specific events occurring on specific entities. The generalized trace has selective features, called "selective tracing" facilities. Different events may be associated to different user treatments, including a standard edition, or ignored.

If an event involves, for instance, a queue connected to a trace function, and a customer connected to another trace function, two user trace functions are connected to the same event, but only one is performed. A standard priority management has been defined, but the user can specify another one to choose between several possible handling functions.

Some complex mechanisms can be represented easily by using the generalized trace. For example:

- In a computer network, when a node buffer is full, the data transmission must be stopped until the buffer has been emptied out.
- In a railway station, the number of opened ticket offices depends on the size of the waiting queue. If the queue is short, some ticket offices are closed. Conversely, if the queue gets too long, some ticket offices open.

These examples can be generalized to the management of minimum and maximum levels for resource allocation. In these two examples, the generalized tracing facility can be used to catch critical events and process them appropriately.

The generalized trace can also be used for tailored applications such as animating a model.

The generalized tracing facilities are descibed in chapter 9, "Tracing facilities".

B.2.2 Generalized statistics

Simulation results are statistics on all the queues defined in the model. Statistics are computed on the service time, the response time, the blocking time, the utilization rate and the number of customers.

The *queue statistics* have been generalized in several ways:

- Basic queue statistics are not restricted to the mean value. The minimum, maximum and variance are also computed automatically.
- Marginal probabilities are not restricted to the number of customers.
- Auto-correlation coefficients can be computed on any result when the regeneration method is used to compute confidence intervals.

To improve simulation time, memory occupation and standard report understanding, statistics can be restricted to a subset of queue and/or results. The "selective statistics" provide the user with facilities to select which statistical results are interesting on which variable. Two selection levels appear: on variables and on statistics.

The minimum set of statistical results is: mean, variance, minimum and maximum values. Then, the user can request marginal probabilities, confidence interval or simulation run length control.

A set of functions is available to specify which statistical results must be computed on which variable. The name of the function specifies the statistical results to compute.

The standard results provided by a simulation concern only queues and classes. Most often, users need other statistical information, such as the network response time. In previous releases, users had to write their own code to collect data and compute statistics. This code was typically embedded in the service descriptions.

In V9.0, the generalized statistics are not restricted to queues. *User statistics* can be computed on any user variable, declared as a *watched* variable, also called "statistical variable". The algorithmic language interpreter tracks any change in the value of the variable and computes the requested statistics.

The generalized statistics are described in section 5.5, "Simulation results".

B.2.3 Accuracy control

QNAP2 can compute confidence intervals on any queue result or statistical variable. The confidence intervals are used to characterize the accuracy of the mean values. When the simulation length increases, the number of samples increases too and the accuracy becomes better.

An approximation can be applied to long simulations: the confidence intervals is approximately proportional to the inverse of the square of the simulated time. A first simulation could provide confidence intervals on the mean value of the variables, from these intervals length, an approximated simulation run length could be computed, by the user, to obtain a specific accuracy on the results, but an automatic mechanism is better.

In V9.0, the user can specify the required accuracy for selected statistical results. The simulation is automatically stopped whend the required accuracy is reached. In order to prevent endless simulations, the maximum simulation run length is specified by the user. The accuracy control mechanism is designed to spare simulation time.

Confidence intervals can be computed on any queue result, as well as on statistical variables, so the simulation run length control facility is available on all model outputs.

The accuracy control mechanism is described in section 5.4.4.2, "Accuracy control".

B.2.4 Exceptions

In QNAP2, communications between the modelling tool and the environment are possible either through file I/O or by calling an external program.

These facilities have been extended to handle asynchronous signals, i.e., interprocess signals, key strokes, real or CPU timers. New mechanisms have been implemented in QNAP2 for exception handling. The exception concept, available in some standard languages or systems, is offered to QNAP2 users through EXCEPTION objects. This facility is extended to standard exception handling.

An exception is connected, on one side, to an external signal, and on the other side, to the corresponding treatment.

In QNAP2, the external signals are defined in a model by a standard name. The availability of this name depends on the software installation and on the host machine, because the operating system must allow users to handle the signal.

When the external signal occurs, a user exception handler is executed before the treatment of the next simulation event.

Exception mechanisms in QNAP2 are also intended to handle some internal conditions, such as the beginning and the end of a simulation. These conditions are connected to standard internal exceptions.

The exception connected to the beginning of the simulation can be used to perform initializations, especially if confidence intervals are computed with the replication method.

Another standard internal exception is connected to the simulation run length control. Its treatment is invoked when the requested accuracy has been reached. The user can display the outputs or resume the simulation, if he hopes a better accuracy.

The exception mechanisms are used especially if QNAP2 is incorporated in a more general environment, to exchange data and orders with other more specific tools. For example:

- Simulation can be used to study the evolution of a railway system. An external user controlled signal generates breakdowns and pauses the simulation. Then, it is up to the user to decide which parameters must be modified, for example: parking or repairing the train, blocking the railway line access. When done, another signal is sent to QNAP2 to resume the simulation.
- Manufacturing systems modelling is often incorporated in a more general environment, including an animation tool, for a better understanding of the system evolution. The animation can be performed on line with new trace facilities. With the exception mechanisms, a user, interested in a particular aspect of the model, can stop the animation, go on event after event, modify interactively some parameters, check the results, or stop the simulation execution.

Exceptions are described in section 4.14, “Exceptions”.

B.3 Analytical solvers

The analytical solvers of QNAP2 provide an effective means to solve models. They provide accurate results with a very small computation time, especially compared to simulation.

Two existing analytical solvers have been extended to handle finite capacity queues and multiple server stations with concurrency sets.

Queueing networks including finite capacity queues can be solved with the CONVOL and MVANCA methods.

Queueing networks including multiple server stations with concurrency sets can be solved with the CONVOL method.

CONVOL is able to solve queueing networks including both finite capacity queues *and* multiple server with concurrency sets, provided that these two features are not used on the same stations.

The analytical solvers are described in chapter 5, "Solvers".

B.4 Algorithmic language

Two enhancements have been performed on the algorithmic language: user-declared functions, and structured identifiers for predeclared procedures and functions.

B.4.1 User functions

With V9.0, the user is now allowed to declare functions. User-declared functions have exactly the same flexibility as user-declared procedures: local variables, arguments passed by value or by reference, forward declarations, generic declarations and references to user functions.

The declaration syntax is exactly the same as for procedures, except that:

- the keyword `FUNCTION` replaces the keyword `PROCEDURE`, and
- the function type must precede the declaration.

Example:

```
/DECLARE/ BOOLEAN FUNCTION ISOK (Q);
                ...
```

User functions are presented in section 2.3.9, "Function declarations".

B.4.2 Hierarchical predeclared procedures and functions

In V9.0, a new syntax has been introduced for predeclared procedure and function identifiers. The identifiers are now structured in the following way: *functionality:keyword*.

The first word refers to a general purpose functionality, e.g., SETSTAT for statistics specifications. The keyword indicates which feature of the functionality is called.

Example:

```
SETSTAT:QUEUE (CPU, DISK)
```

This procedure call requests that standard queue statistics be computed on the CPU and DISK stations.

The major functionalities that now have structured identifiers are the following:

- Statistics specification: SETSTAT:*keyword* procedures.
- Result access functions: GETSTAT:*keyword* functions.
- Timer control: SETTIMER:*keyword* procedures.
- Exception control: SETEXCEPTION:*keyword* procedures.
- Simulation control information: GETSIMUL:*keyword* functions.
- Generalized trace specifications: SETTRACE:*keyword* procedures.
- Traced event information: GETTRACE:*keyword* functions.

B.5 Compatibility with V8.0

QNAP2 V9.0 is compatible with V8.0. All the functionalities of V8.0 are available in V9.0.

Some new functionalities replace older ones. The new ones are designed to be more powerful than the old ones. The old functionalities still exist in V9.0, but they might be suppressed in the future. Users are encouraged to use the new features.

The obsolete features are the following:

- /CONTROL/ parameters for statistics specification should be used only with analytical solvers.

  With the discrete event simulator, most /CONTROL/ parameters are overridden by SETSTAT:*keyword* procedures. Mixing /CONTROL/ parameters with SETSTAT:*keyword* procedures is not recommended, as the /CONTROL/ parameters will probably be ignored (a warning message will tell you).
- The SETTRACE procedure of V8.0 is now replaced by SETTRACE:BOUNDS and SETTRACE:WIDTH. Using the old SETTRACE procedures yields a syntax error.

New features V9.1, V9.2 & V9.3 $\boxed{\text{C}}$

This chapter presents the new features of QNAP2 releases 9.1, 9.2 & 9.3.

C.1 New features of QNAP2 V9.1

These features are presented in the QNAP2 User's Guide and in the QNAP2 Reference Manual.

1. Multiple requests and-or releases of pass grants to semaphores or resource units.
   See `PMULT` and `VMULT` procedures.
2. The `CONVERT` function is now able to convert QNAP2 internal names into `STRING` variables. (see the reference manual)
3. The `HOSTSYS:GETENV` function can get the value of environment variables. Error codes are returned by the `HOSTSYS:GETERCOD` function. (see the reference manual)

C.2 New features of QNAP2 V9.2

These features are presented in the QNAP2 User's Guide and in the QNAP2 Reference Manual.

1. The simulation execution profile.
   This feature is available only on UNIX platforms. See the reference manual (`GETPROFILE` and `SETPROFILE`).
   Example :

```
/EXEC/
BEGIN
  SETPROFILE:CLEAR;                  & Clear profiling structures
  SETPROFILE:METERPROC(procedure);   & Ask for results on procedure
  SETPROFILE:STARTMETER;             & Start profiling the execution
  SIMUL;
  SETPROFILE:STOPMETER;              & Stop profiling
  IF GETPROFILE:ISMETERED(procedure)
    PRINT("Consumed time : ",        & Print results
          GETPROFILE:RESULTS:TOTALCPU(procedure));
  SETPROFILE:CLEAR;
END;
```

2. The parallelization of replications.
   See its description in chapter "Solvers" (Estimation of confidence intervals).
3. The SPLITMAT solver.
   See its description in chapter "Solvers" (The Split-Match approximation solver SPLIT-MAT).

C.3 New features of QNAP2 V9.3

These features are described in this addendum.

1. Running QNAP2 using the command line.
2. Suppression of QNAP2 graphic commands.
   The QNAP2 graphic commands are not supported any longer. See how to emulate them in section "Graphics and QNAP2".
3. Documentation on error messages.

4. New statistical functions.
   See their descriptions in the Reference Manual (`GETSTAT:SAMPSIZE`, `GETSTAT:SAMPTIME`, `GETSTAT:THRUPUT:MEAN`, `SETSTAT:THRUPUT:MEAN`).
5. New traced events.
   See their descriptions in the Reference Manual ("`PMULT`" and "`VMULT`" events).
6. The `HOSTSYS:SHELL` function can issue a shell command.

Reserved Identifiers $\boxed{\textbf{D}}$

The following are reserved words in the QNAP2 system. Using them as an identifier yields a compilation error.

ALL	FOR	NOT	UNTIL
AND	FORWARD	OBJECT	VAR
ANY	GENERIC	OR	WATCHED
BEGIN	GOTO	REF	WHILE
DO	IF	REPEAT	WITH
ELSE	IN	STEP	
END	IS	THEN	
FALSE	NIL	TRUE	

The following are predeclared identifiers in the QNAP2 system. It is possible to use them as user identifiers in the following cases:

- as an object attribute,
- as a local variable in a procedure or function.

However, this is not a recommended programming practice.

ABORT	FIX	MIN	SETTMAX
ABS	FLAG	MOD	SETTRACE
ACOS	FLINKCUS	MODIFY	SHBREAK
AFTCUST	FLISTCUS	MOVE	SHINSERT
ASIN	FREE	MRESPONSE	SIMACCUR
ATAN	FSYSERR	MSERVICE	SIMSTART
AUDIT	FSYSGET	MTHRUPUT	SIMSTOP
BEFCUST	FSYSINPU	NB	SIMUL
BLOCK	FSYSLIB	NBAUDTED	SIN
BLOCKED	FSYSOUTP	NBAUDTOR	SKIP
BOOLEAN	FSYSPRIN	NBIN	SOLVE
BREAK	FSYSTRAC	NBOUT	SONNB
CANCELBR	FUNCTION	NETWORK	SQRT
CARD	GET	NEW	STARTED
CBLOCKED	GETCPUT	NORMAL	STATE
CBUSYPCT	GETDATE	OPEN	STOP
CCLASS	GETDTIME	OUT	STP
CCUSTNB	GETLN	OUTPUT	STRING
CHARCODE	GETMEM	P	STRLENGTH
CHREXCLUDE	GETSIMUL	PAUSE	STRMAXL
CHRFIND	GETSTAT	PCUSTNB	STRREPEAT
CLASS	GETTRACE	PLOATT	SUBSTR
CLEARSCR	GO	PLOCUR	TAN
CLOSE	HALT	PLOHIS	TIME
CONVERT	HEXP	PLOSEC	TIMER
COS	HISTOGR	PMXCUSTNB	TRANSIT
COX	ICLASS	PRINT	TRANSLATE
CPRIOR	INCLUDIN	PRINTF	TSYSPERI
CQUEUE	INDEX	PRIOR	TSYSSPCT
CRESPONSE	INSERT	PROCEDURE	TSYSTMAX
CSERVICE	INSERVICE	QUEUE	TSYSTRAC
CST	INTEGER	RANDU	TSYSTSTR

CTHRUPUT	INTREAL	REAL	TSYSWDOG
CURVE	INTROUND	REALINT	TYPNAME
CUSTNB	ISAUDTED	REFSON	UNAUDIT
CUSTOMER	ISBLOCK	REMOVE	UNBLOCK
CVNOERR	JOIN	RESET	UNIFORM
DELETED	JOINC	RESTORE	UNSET
DISCRETE	LAST	RESULT	UTILITY
DISPOSE	LINK	REVERSE	V
DRAW	LIST	RINT	VCUSTNB
DUMP	LOG	RTRIM	VRESPONSE
ERLANG	LOG10	SAMPLE	WAIT
EXCEPTION	LTRIM	SAVE	WAITAND
EXP	MAP	SAVERUN	WAITOR
EXPO	MARKOV	SERVNB	WATCHED
FATHER	MAX	SET	WRCHCODE
FAUDITED	MAXCUSTNB	SETBUF	WRITE
FAUDITOR	MBEGIN	SETEXCEP	WRITELN
FILASSIGN	MBLOCKED	SETRETRY	WRSUBSTR
FILE	MBUSYPCT	SETSTAT	
FILSETERR	MCUSTNB	SETSYN	
FIRST	MENDGR	SETTIMER	

Index

Index